

AD-A124 872

AN INTERACTIVE AND AUTOMATED SOFTWARE DEVELOPMENT  
ENVIRONMENT(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING S M HADFIELD DEC 82

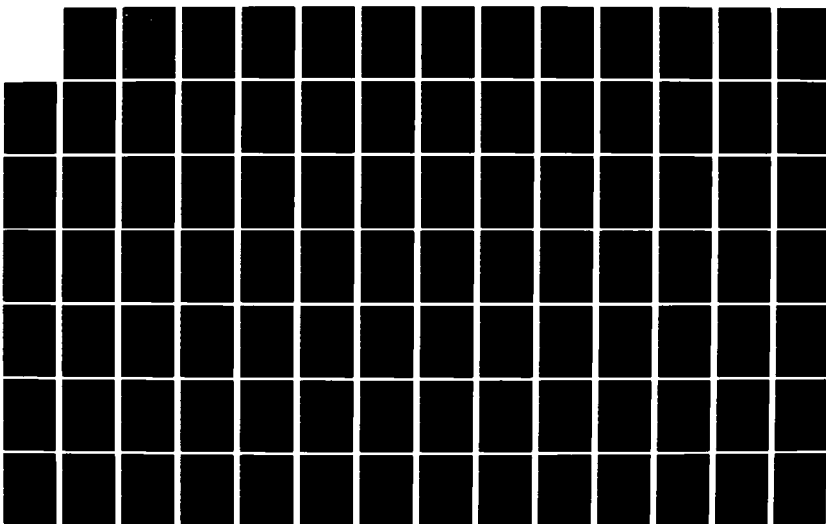
1/4

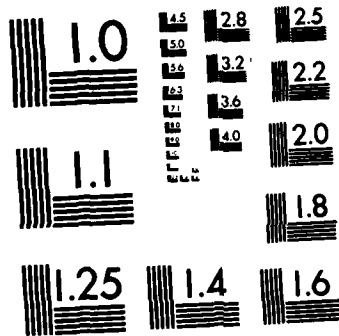
UNCLASSIFIED

AFIT/GCS/EE/82D-17

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

AD A124872



AN INTERACTIVE AND AUTOMATED  
SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

AFIT/GCS/EE/82D-17 Steven M. Hadfield  
2Lt USAF

DTIC FILE COPY

DTIC  
ELECTE  
FEB 24 1983  
S  
E

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY (ATC)  
**AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

83

02

022 061

This document has been approved  
for public release and sale; its  
distribution is unlimited.

AFIT/GCS/EE/82D-17

AN INTERACTIVE AND AUTOMATED  
SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

AFIT/GCS/EE/82D-17 Steven M. Hadfield  
2Lt USAF

Approved for public release; distribution unlimited



AFIT/GCS/EE/82D-17

AN INTERACTIVE AND AUTOMATED  
SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

Presented to the Faculty of the School of Engineering  
of the Air Force Institute of Technology

Air University  
in Partical Fulfillment of the  
Requirements for the Degree of  
Master of Science

by  
Steven M. Hadfield, B.S.  
2Lt USAF

Graduate Computer Science  
December 1982



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Dist: _____	
Avail: _____	
Dist: _____	
A	

Approved for public release; distribution unlimited

## PREFACE

This report is a result of my effort to accomplish a high level design and initial implementation of an automated and interactive software development environment. The resulting implementation of this thesis investigation is an easy to use, yet very powerful aid for the development of software in accordance with accepted software engineering principles. However, this current implementation is only a partial realization of the carefully developed design specifications for an eventual environment involving a higher level of sophistication. The last chapter of this report outlines a progression of follow-on efforts required to accomplish the eventual and complete realization of the environment. My hope is to encourage the continued development of this software development environment, formally identified as the "Software Development Workbench".

I wish to express my sincere appreciation to Dr. Gary B. Lamont, the advisor of this investigation, for his professional guidance, insight, and patience throughout the duration of this effort and to Ric Mayer, Program Manager of the Integrated Computer Aided Manufacturing/Systems Engineering Methodologies program, for his sponsorship and direction. I also wish to thank the members of my thesis committee, Major Hal Carter, Ray Rubey, and Major Michael Varrieur, for their support and patience.

Finally, I desire to dedicate this document, as well as the blood and sweat behind it, to my father, who's personal integrity and continuous support have been a constant source of inspiration and guidance for me.

Steven M. Hadfield

## Contents

	Page
	----
Preface	ii
List of Figures	vii
Abstract	ix
1 Introduction	1
1.1 Thesis Objective	2
1.2 Background	2
1.3 Problem Statement	15
1.4 Scope of the Thesis	17
1.5 Assumptions	17
1.6 Approach	18
1.7 Summary	21
2 Requirements Definition	22
2.1 Introduction	23
2.2 Model of the Existing Software Development Process	25
2.3 SDW Objectives and Concerns	31
2.4 Functional Model of the Software Development Workbench	48
2.5 SDW Evaluation Parameters and Criteria	82
2.6 Summary	85
3 Preliminary Design	87
3.1 Introduction	88
3.2 Evolutionary Design Strategy	91
3.3 SDW Configuration Model	95
3.4 Resolution of the SDW Development Objectives and Concerns	101
3.5 SDW Structure Chart Model	120
3.6 Summary	128
4 Detailed Design	130
4.1 Introduction	131
4.2 SDW Component Selection	132
4.3 Detailed Design of the SDW Executive	144
4.4 Design of the Project Data Bases	173
4.5 Summary	160

5	The Implementation Stage	182
5.1	Introduction	183
5.2	The Choice of an Implementation Language for the SDWE	186
5.3	The SDW Implementation Strategy	190
5.4	SDWE Implementation Specifics	192
5.5	SDWE Update to Version 1.1	199
5.6	Summary	203
6	Integration of the Software Development Workbench	205
6.1	Introduction	206
6.2	Installation of the SDW Components	206
6.3	Integration of the SDW Components and the SDWE	207
6.4	Installation of the SDW on the Central ICAM Development System	209
6.5	Summary	210
7	Operations and Maintenance of the Software Development Workbench	211
7.1	Introduction	212
7.2	Development of the SDW Documentation Package	212
7.3	Maintenance Activities on the SDW	213
7.4	Evaluation of the Software Development Workbench	215
7.5	Summary	217
8	Conclusion/Recommendations	218
8.1	Introduction	219
8.2	Design Summary	219
8.3	Implementation/Test Strategy	221
8.4	Recommendations for Future Investigations	222
	Bibliography	225
	Appendix A: A Model of the Existing Software Development Process	232
	Appendix B: SDW Data Dictionary	255
	Appendix C: Specification of Preliminary Design Modules	265

Appendix D: Detailed Requirements Definition for the SDWE	285
Appendix E: Preliminary Design for the SDWE	294
Appendix F: Algorithmic Design of the SDWE	311
Appendix G: Listing of SDW Command Codes	319
Appendix H: SDWE File Descriptions	323
Appendix I: SDWE User's Manual	336
Appendix J: SDWE Installation Guide	347
Appendix K: SDWE Maintenance Guide	352
VITA	365

# LISTING OF FIGURES

Figure Number	Title	Page
1	Software Life-Cycle	5
2	SADT Activity Box	28
3	Data Flow Diagram Constructs	48
4	SDW Functional Model Outline	51
5	SDW Functional Model: Top Level	52
6	Perform Software Life-Cycle	53
7	Perform Requirements Definition	57
8	Develop Draft Requirements	59
9	Translate Requirements into a Machine-Readable Form	61
10	Develop Preliminary Design	63
11	Develop a Draft Preliminary Design	65
12	Validate Preliminary Design	67
13	Develop Detailed Design	69
14	Implement and Test Software System	71
15	Convert to Syntactically Correct Code	73
16	Test Code with Traces and Error Handling	75
17	Optimize the Code	77
18	Integrate to and Validate on Target Machine	79
19	Maintain and Operate Software System	81
20	Tool Variety Progression Plan	93
21	Tool Integration Progression Plan	93

22	SDW Configuration Model	96
23	Sample HIPO Function Chart	121
24	IPO Diagram Sample	122
25	HOS Function Specification	123
26	SDW Structural Model	126
27	A-0 Utilize the SDW	149
28	A0 Utilize the Software Development Workbench	150
29	A1 Initialize the SDW	151
30	A4 Execute the User's Command	153
31	A41 Provide Functional Tool Group	155
32	A42 Provide Help Facilities	156
33	A43 Access the Pre-Fab Software Description Data Base	157
34	SDWE Preliminary Design Model-1	164
35	SDWE Preliminary Design Model-2	165
36	Project Data Base Design	179
37	Preliminary Design Top Level	185



## ABSTRACT

The purpose of this investigation is to ~~1)~~ define both the detailed requirements and the preliminary design for an automated and interactive software development environment, and ~~2)~~ develop an initial implementation of that environment. The specified requirements for this environment emphasize the need to support the entire software life-cycle as a continuous and iterative process. In particular, the concepts of integration, traceability, flexibility, and user-friendliness are accentuated. The preliminary design delineates the high level design specifications, configuration schemes, and generic tool categories with which the previously mentioned requirements may be satisfied.

Detailed designs are developed for the integrating interface/controller sub-system and the development data storage scheme for the initial implementation of the environment. The interface/controller sub-system has been implemented and tested using the DEC Command Language (DCL) and PASCAL. This sub-system is integrated with an initial software development tool set executing on the VAX-11/780 computer using the VMS operating system. This initial implementation, called the Software Development Workbench (SDW), is an extremely effective and easy to use aid for extending the cognitive and notational powers of the software developer.

## Chapter 1: Introduction

### 1.1 Thesis Objective

The objective of this thesis investigation is to perform the initial development and implementation of a software development environment for the Air Force Institute of Technology (AFIT). This software development environment is entitled the Software Development Workbench (SDW). The SDW supports the development and maintenance of software from conception to termination by using automated and interactive tools that apply the principles of software engineering.

### 1.2 Background

A software development environment is an integrated set of automated and interactive software development tools that aid the software engineer to develop quality software products and documentations. The software products and documentations that are developed with the use of a software development environment include requirements definitions; design specifications; source and executable program codes; test plans, procedures and results; as well as other associated documentation such as guides and manuals for operations and maintenance of the software.

A well planned and implemented software development environment can effectively assist in the development of reliable and maintainable computer software. The typical software development environment includes both hardware and software tools to aid the software designer/programmer in the production of software. Software development environments may consist of a minimal set of primitive tools, such as editors and compilers, that support only the actual coding of software. However, the most effective environments are those with extensive sets of powerful tools that support the most modern state-of-the-art methodologies for dealing with software from its very conception through its eventual termination (Ref 14). The methodologies that are supported by such environments are a result of investigations in software engineering.

During the past two decades, the discipline of software engineering has developed in response to increasing problems with the production and maintenance of computer software. In particular, the goals of software engineering have been to improve the software production process and software quality. The concept of a software life-cycle has been defined by many different authors in many different ways. The version of the software life-cycle that is popular with the AFIT software community is composed of six stages (Ref 40:4). This life-cycle is illustrated in Figure 1. The stages of this life-cycle are requirements definition,

preliminary design, detailed design, implementation (coding), integration, and maintenance. The verification and validation stage, that is included in many versions of the life-cycle (Ref 59), is left out of this version. There are two important reasons for this. First, the term, "verification", is not really used correctly in this context. According to the New American Webster Dictionary, the word "verification" means "(the act of)... proving something to be true" when, in fact, software products are seldom ever proved to be true. The term "validation", which means the "supporting of something's validity by facts", is a much more accurate expression of the actual objectives of software testing.

The reason why the validation and testing activities are not included as a formal stage in the software life-cycle is that these activities should actually take place throughout the entire life-cycle. The definition of the requirements should be tested both internally and against the needs of the user. Likewise, each of the other stages should be tested both internally and against the

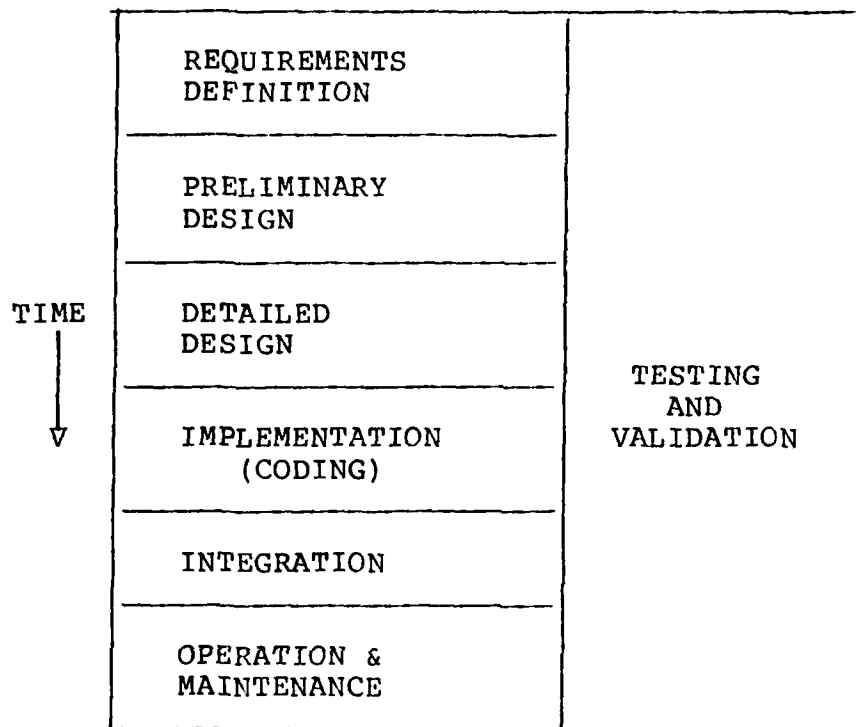


Figure 1: Software Life-Cycle

products of the earlier stages. Each of the life-cycle stages have their own objectives and the fulfillment of those objectives are fundamental to the progression of the development into the next stage.

The objective of the requirements definition stage is to formulate an explicit statement of what the proposed software system must do. The emphasis of this stage is on the what the system is to accomplish and not on how the system will accomplish it. Careful attention should be paid

not to constrain the system by specifying system mechanics (the how) during the requirements definition stage. Several activities provide means for achieving this objective. Scoping involves limiting of the objectives to be accomplished so that the problem statement actually addressed is solvable with current technology and available resources. Needs analysis refers to the careful study of the user's needs for the software system. Often an "As-Is" System Definition of how the problem is currently being addressed helps in formulating the requirements. A certain level of conceptual design is also useful in determining if the stated requirements are actually feasible. The main activity of the requirements definition stage is the development of a functional model that states the exact functions that must be accomplished with the system. This functional model is developed from the user's point of view and is usually in terms of data flows and functions on these data flows.

The mechanisms of the system structure begin to be defined during the preliminary design stage. During this stage, the particular requirement specifications of the previous stage are allocated to specific design components. The components may be further specified and broken down into sub-components if needed. Thus, a hierarchical description of the software system is formed. Within this hierarchical description or framework, components and sub-components are

realized in terms of functional modules. The functional modules are, at this stage, just black boxes with defined inputs and outputs. In many applications, a hierarchical implementation and test plan is begun to be formulated during this stage. This plan uses an "incremental" approach to implementation and testing which uses the hierarchical structure and facilitates easier system validation.

Each of the components of the preliminary design are further specified in the next stage referred to as detailed design, or algorithmic design. During this stage, the black-box descriptions of the functional modules are translated into specific algorithms. Many times some of the functional modules can be satisfied by existing software packages. Test plans and data for each of the algorithms are often developed and incorporated into the implementation and test plan during the detailed design stage.

The implementation stage is the actual coding of the software system in a particular programming language. Testing is incorporated into this stage in accordance with the implementation and test plan.

The goal of the integration stage is to integrate the software system code to the target hardware and to perform the independent testing of the software system. During the development of the software system, there is often an independent project that is responsible for validating the



final software product. The use of independent testing and validation is especially effective because of its objectivity. An independent testing agency looks specifically at how well the software system meets the specified requirements. This manner of testing most often employed is called "Black-Box Testing" because it specifies certain inputs, runs the system, and compares the actual output with the anticipated output. The mechanics of the system are not of importance during this type of testing, thus the test planning can be concurrent with system development.

The sixth stage of the software life-cycle is that of operation and maintenance. Operation of the software system involves the actual usage of the system by the users. During this time errors may be detected in the software, or needs for changes in the required functions of the software may be realized. The modifying of the software system to resolve errors or additional user requirements is known as software maintenance. In the event of these occurrences, software must be modified and this is known as maintenance.

In order to support these phases, many methodologies and tools have been developed. However, these tools often require a great deal of bookkeeping, illustrating, and consistency checking. Many of the contemporary methodologies have been automated. Thus the massive

potential of the computer can be realized. Automation relieves the burden of the many tedious tasks associated with the methodologies. In fact, numerous automated tools presently exist to support each of the phases of the software life-cycle (Ref 40:1-5). However, almost all of these tools are disjoint, and they often do not interface to tools of the other phases of the life-cycle.

A software development environment is a collection and integration of automated software development tools that should adequately support the entire software life-cycle. The universe of potential development environments can be realized as a two dimensional space with one dimension being the number and variety of tools and the other dimension being the level of integration achieved between the tools. Most contemporary software development environments have emphasized only one of the these two dimensions (Ref 67:2). Those environments that utilize a "tool kit" approach view the concept of an environment as a collection of many automated tools that are used disjointly. The "job/union shop" approach defines an environment to be a limited collection of tools that are integrated in accordance with a single development philosophy. As one would expect, the optimal environment for most applications is found by extending in both dimensions of the universe potential. This requires an environment that has many tools that may be used separately or integrated to support a life-cycle

methodology. This approach to a software development environment provides both the flexibility of the "tool kit" approach and the integration of the "job/union shop" approach.

The concept of a Software Development Workbench (SDW) is a broader perspective that defines not only the individual tools that are to be incorporated and how they interface to one another, but also specifies the physical components required for the environment. That is, a SDW defines the tools and the framework within which the tools are used.

Up to the present time, the emphasis on automated software development environments has been on the development of specific tools or development of environments tailored to a specific and narrow domain of application. This statement is substantiated by a noted expert on the subject of software development environments, Leon Osterweil of the University of Colorado.

"Most current tools and tool systems focus their support on narrow aspects of the software processes, such as editing and testing, and ignore the other areas." (Ref 60:35)

He elaborates further on the fact that environments should be developed that support the entire life-cycle and that the issues of integration of the specific tools and ease of

teachability and use should be of the utmost concern. Furthermore, he challenges that:

"Although these environments and their benefits have been widely discussed, there has been relatively little research or actual implementation in this area." (Ref 60:35)

Osterweil is quite correct in his statement that there has been a great deal of discussion on the subject. A primary focus of much of this discussion has been the fundamental concerns to be realized in the establishment of a software development environment. The list of concerns resulting from these discussions is lengthy, yet nevertheless its content is fundamental to the understanding of the objectives of a software development environment. The concerns include the concepts of integration, user-friendliness, life-cycle support, flexibility, consistency, traceability, explicitness, documentation capabilities, testability, and the capability of updating (Ref 60:36-37). These concerns are discussed in the chapter on requirements definition.

One example of an on-line environment that address these fundamental concerns is the UNIX\* Programmer's Workbench (Ref 40:345-357). A product of Bell Laboratories, the Programmer's Workbench (PWB) is built to operate on the UNIX\* operating system. The Programmer's Workbench provides

for uniform program development and supports remote job entry, source code control/modification, documentation preparation and other tasks. The PWB supports the production of software that will be compiled and run on non-UNIX\* target systems. As a result of being built on the UNIX\* operating system, the environment achieves some successful integration by utilizing the common command syntax and generic file structure of the operating system. The PWB does, however, lack capabilities for requirements analysis and specification, quality assurance, and specific software methodologies (Ref 40:356).

The design philosophy behind the PWB is to get the users on the system as soon as possible and let their needs and experiences drive the design. The designers of the PWB believe in building software quickly and throwing much of it away. Many small programs are preferred to a few large ones and a monitor is used to tract and log user problems. The UNIX\* PWB possess a fairly high degree of integration and is a prime example of the "union shop" approach to development environments (Ref 40).

Another interesting and contemporary approach to developing a Software Development Environment is the Ada Programming Support Environment (APSE). This environment is much more of a "tool kit" approach than was the PWB. The requirements for the APSE are stated in the "STONEMAN"

requirements (Ref 28). The APSE is an excellent example of a language-oriented environment. There are actually two government contracts to build APSEs, one with the U.S. Army, called the Ada Language System (ALS) and one with the U.S. Air Force, called the Ada Integrated Environment. The APSE utilizes a kernal host dependent module for low level I/O, user interfaces, program execution, and a data base, with the majority of the environment being host independent. The APSE is primarily an implementation and integration oriented environment that provides extensive separate compilation facilities, configuration managers, and the addition of simulators and testbeds. However, the APSE also lacks capabilities for requirements analysis and specification and specific software design methodologies.

The shortcomings with both the PWB and the APSE is that they lack facilities for many of the pre-implementation software development activities. A study done by TRW found that sixty-four percent of the errors encountered in a range of software projects could be traced back to the requirements definition and design phases of the project (Ref 86). Thus, there is quite a legitimate case for a major emphasis to be placed on these phases of the life-cycle. This was exactly the concern during the development of the Systems Requirements Engineering Methodology (SREM), which was developed for the Army's Ballistic Missile Defense Advanced Technology Center (Ref 3).

SREM utilizes the System Specification Language (SSL) to state requirements in a machine readable form. Automated tools called the Requirements Evaluation and Verification System (REVS) and an extended version called EREVS are used to check requirements stated in RSL for consistency and completeness, and then produce graphics called R-nets, which are process flow diagrams. Although the present state of SREM deals almost exclusively with requirements definition and preliminary design, major efforts are being initiated to extend the system to a life-cycle supporting environment.

All of the efforts to develop a software development environment are major scale projects with on going objectives and difficult philosophical and methodological decisions to be made. Osterweil capsulized this fact in his statement:

"The task of creating effective (development) environments is so difficult because it is tantamount to understanding the fundamental nature of the software process." (Ref 60:36)

The sponsor of this investigation is the Integrated Computer-Aided Manufacturing/System Engineering Methodology (ICAM/SEM) group. They are presently involved with a 15 man-year project called the Integrated Systems Development System (ISDS). This system is a large scale development environment for the development of not only software

systems, but all types of manufacturing systems. Integration in the ISDS is achieved through a shared database called the Common Data Module (CDM). The ISDS uses an approach to the development environment that extends along both dimensions of the universe of potential for software development environments. Many types of tools are used in ISDS and yet a high degree of integration is to be imposed by the use of the Common Data Module.

### 1.3 Problem Statement

The Air Force Institute of Technology (AFIT) and the Air Force software community as a whole have a great need for a life-cycle oriented software development environment. AFIT's software community is continuously developing major software products during thesis investigations and other Air Force sponsored research. The objective of this thesis effort is to define AFIT's requirements for a software development environment and then design and implement a prototype Software Development Workbench (SDW) to satisfy these requirements. The SDW is required to support three categories of user's within the AFIT software community. These categories are listed below.



- 1) Students enrolled in the AFIT Software Engineering course.
- 2) Students and faculty involved in M.S. and Ph.D. research.
- 3) Students and faculty involved in other software related activities.

The target and development machine for the SDW development effort is the Digital Equipment Corporation's VAX 11/780 utilizing the VMS operating system. This machine and operating system are chosen for three primary reasons, 1) it is available in the AFIT/Digital Engineering Laboratory, 2) it is the target machine for the ISDS development, and 3) many existing automated tools exist in VAX/VMS compatible versions.

The objectives of this investigation are not limited to the development of a software development environment for AFIT. The sponsoring ICAM/SEM office is greatly concerned with the issue of tool integration as it applies to the ISDS Common Data Module. The rehosting of tools for the VAX-11/780 computer provides ICAM/SEM with greater flexibility in utilizing these tools on their nation-wide computer network that will eventually become the ISDS (Ref 58).

#### 1.4 Scope of the Thesis Investigation

As pointed out in the background section of this chapter, the development of a software development environment is a very involved effort. This thesis investigation effort is a first step towards the realization of such an environment for the AFIT software community. The requirements definition of the prototype workbench deals with the identification of functions (tools) and scenarios (methodologies) to be supported by the environment. The requirements definition stage also emphasizes the defining of the major objectives and concerns that must be satisfied by the design. Functions present in the stated requirements that are already available in contemporary systems, such as REVS, acquired through the sponsoring ICAM/SEM office. The prototype design consists of a preliminary design of the workbench, and a detailed design of the SDW executive and some other needed tools. Implementation is limited to the coding of the SDW executive and the re-hosting of available tools.

#### 1.5 Assumptions

The implementation of the Software Development Workbench prototype is done on the VAX-11/780 computer, because of the wide availability of software development

tools on this system, as well as the availability of the AFIT/DEL VAX-11/780 as a development computer.

The sponsoring ICAM office had promised to provide tools, under their control, to the Software Development Workbench effort.

### 1.6 Approach

This thesis investigation begins with an extensive search and review of literature dealing with software engineering and software development environments. In particular, this research deals with gaining a thorough understanding of the software life-cycle, the methodologies used to support the life-cycle, and how to improve life-cycle activities using an automated and interactive development environment. The review of this literature provides a "sound" background for the development of a software development environment.

Utilizing the understanding gained from the literature review, the thesis investigation moves into the requirements definition stage of development. The beginning of this stage introduces and justifies the methodologies used to accomplish the objectives of this stage. Then, a model of the existing software life-cycle is developed. Following this analysis of the existing life-cycle, a comprehensive

explanation of the objectives and concerns fundamental to the development of a software development environment is delivered. These objectives and concerns are the summarization of those identified in the review of the literature. Using the model of the existing software life-cycle and the summarization of development environment objectives and concerns, the high-level requirements for the Software Development Workbench are defined. These requirements take the form of a model of the software life-cycle as it should be supported by a software development environment, such as the Software Development Workbench. Finally, a set of evaluation parameters and criteria is established to provide for the judging of how well the SDW implementation meets the stated requirements.

The preliminary design stage of the SDW development effort is broken down into four sections. The first section establishes an Evolutionary Design Strategy for the continuing development of the SDW. The next section develops the SDW Configuration Model that identifies the major systemic components of the SDW and how they are connected. The third sections explains how each of the objectives and concerns of the SDW development are resolved. Finally, the fourth section uses Structure Charts to portray the hierarchical framework of the software components of the SDW. The resulting SDW Preliminary Design defines the structure of the initial SDW and provides a guideline for

the future development of the SDW.

The next stage in the SDW development is the detailed design stage. This stage involves the detailed development of a SDW Executive which is to be used as a top-level interface to and controller of the SDW. The component tools and capabilities of the initial SDW are selected. A specific capability for a data base of existing software products is also delineated. Finally, the initial schema is established for the SDW data bases that hold the development data.

Implementation and testing of the initial version of the SDW deals with the coding and validation of the SDW Executive sub-system that controls the workbench. Also, the component tools specified in the detailed design are loaded onto the AFIT VAX 11/780.

The SDW Executive is integrated to the other SDW components during the integration stage. The interfaces between all of these components are also tested at this time.

The final stage of the initial SDW development effort is the operations and maintenance stage. The objectives of this stage are to operate and test the SDW as an operational environment. This involves the training of the SDW users and the resolution of problems found in the system by the

users.

### 1.7 Summary

The development of the Software Development Workbench is an effort to introduce an interactive and automated capability for the production of computer software. The SDW is developed utilizing state-of-the-art software engineering techniques to support all the stages of the software life-cycle. The SDW development is geared to supporting the AFIT software engineering courses and the many thesis and other software developments with the AFIT software community.

## CHAPTER 2: REQUIREMENTS DEFINITION

## 2.1 Introduction

Requirements Definition is the complete and explicit statement of the problem to be solved. Usually this stage is achieved with a great deal of interaction with the user of the software system. The end result of this stage is a Requirements Definition Document that uses graphical and textual means to unambiguously state the problem to be addressed. A complete, explicit, and unambiguous statement of the system requirements is the goal of the requirements definition stage. The primary component(s) of the Requirements Definition Document is a functional and/or data model of the proposed system. These models define the functional/data specifications for the system. The Requirements Definition Document may also include a description of the fundamental objectives that must be achieved by the system and the concerns that guide the development of the system. A specification of the target environment may be included in the Requirements Definition Document. If the proposed system is to be used by a variety of users, a description of how each type of user is to view the system is needed in the Requirements Definition Document. Often a set of evaluation parameters and criteria is included in the Requirements Definition Document to assist in the testing of the system to meet its specified



requirements. The content of the Requirements Definition Document are highly dependent on the nature of the proposed system. However, a great deal of care must be taken to insure that the document produced specifies the system in enough depth and from enough viewpoints to allow for its proper development.

The purpose of this chapter is to develop the system requirements for the Air Force Institute of Technology's Software Development Workbench (SDW). However, the system requirements for the SDW are not just specific to the AFIT software community but must also reflect the needs of the ICAM/SEM users. First a careful analysis of the generalized software development process is performed. This is done in order to establish a sound understanding of the problems to be addressed by the SDW. A comprehensive set of objectives and concerns fundamental to the SDW is listed and explained. With this background established, the specific high-level requirements for the SDW are defined. Contemporary software methodologies are utilized to describe the existing software development process and define the SDW requirements. These methodologies employ easy to understand two-dimensional graphical techniques and are examples of the types of methodologies to be supported by the SDW. Finally, a set of evaluation parameters and criteria is established to aid in assessing the extent to which the SDW development fulfills its requirements.

Numerous software engineering methodologies have been developed to facilitate the explicit definition of requirements (Ref 90). Most of the methodologies used for describing requirements utilize some type of graphic medium. The primary reason for this is that the requirements must be easily understandable. Graphic techniques are more easily understood in most instances. Ideally, the requirements are used as a means of communicating with the user to insure the designer's concept of what is needed is identical to that of the user's. Graphic techniques assist those unfamiliar with the rigor of the software community's dialect to quickly comprehend what the designer has in mind. The methodologies used for the SDW development are described in generality to allow the reader to follow the discussion of this chapter. Much of the detail inherent in these methodologies is omitted since it is not fundamental to the understanding of these applications of the methodologies.

## 2.2 A Model of the Existing Software Development Process

Often, the problem to be solved by a proposed software system is already being addressed by some other system. The development of the proposed software system is a result of inadequacies in this existing system. A careful analysis of this existing system is often necessary before the statement of requirements for a new system can be established. This

analysis of the existing system is facilitated by the development of an "As-Is" model. This "As-Is" model describes how the problem to be solved by the new system is presently being addressed. The understanding gained from the "As-Is" model helps the designer to develop the requirements and even the design of the new system.

The ICAM/Systems Engineering Methodologies (SEM) Group realizes the importance of an "As-Is" model. The ICAM Systems Life-Cycle (Ref 53:334) formalizes the development of an "As-Is" model as part of its initial system development stage called "Needs Analysis". This needs analysis stage precedes the requirements definition stage. The objective of the needs analysis stage is to establish a formal statement of the user's needs for the new system. The development of an "As-Is" model is a primary vehicle for achieving the needs analysis objective.

Prior to the developing of the requirements for the Software Development Workbench, an "As-Is" model of a generic view of the software development process is created. This model describes all of the stages that a software system encounters, from conception to termination. Specific areas within the model that require the application of automated interactive tools are identified.

This "As-Is" model of the software development process is derived from a model of the Manufacturing Systems Development Life-Cycle developed by the Control Data Corporation for the Integrated Systems Development System (Ref 58).

Three software engineering techniques are analyzed for possible use in defining the "As-Is" model of the software development process. They are the Softech Structured Analysis and Design Technique (SADT) (Ref 79), the Data Flow Diagram technique (Ref 90:49), and the IBM Hierarchical Input Process Output (HIPO) technique (Ref 90:50).

The first technique analyzed is the Softech SADT. This methodology utilizes two separate types of diagrams to illustrate the proposed system. The "Activity Diagram" uses labeled boxes to represent activities and labeled vectors to represent input data flows, output data flows, control flows, and mechanisms. The side of an activity box to which a vector is attached identifies whether it is input, output, control, or mechanism. The Figure 2 illustrates how the vectors are identified.

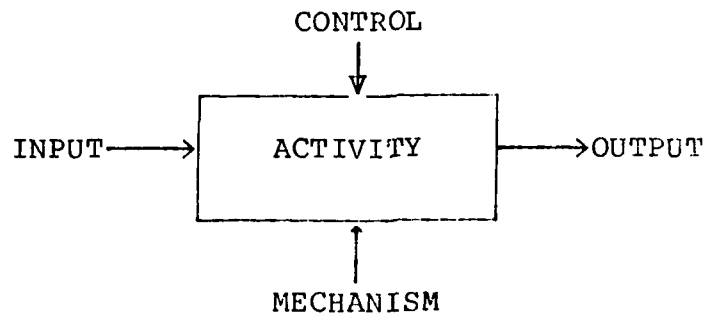


Figure 2: SADT(TM) Activity Box

Each of the SADT Activity Diagrams has one or more Activity Box. Activity boxes may be described by a separate Activity Diagram. By describing Activity Boxes with Activity Diagrams, a hierarchical structure of increasing detail is constructed.

The other type of diagram in the SADT methodology is the Data Diagram. These diagrams are very similar to the Activity Diagrams except the boxes now represent data items and the vectors are activities involving the data. The Activity Diagrams view the proposed system from a functional (procedural) viewpoint, while the Data Diagrams perceive the system as a collection and transformation of data items. Ideally, both types of diagrams are used in order to facilitate multiple perspectives on the system. However, in practice, usually just one of type of diagram is used because of time and other constraints. The Activity

Diagrams are usually chosen over the Data Diagrams. One justification behind this may be that the Activity Diagrams more closely resemble the contemporary concept of a "Black Box". This concept views a system as a set of inputs (data items, controls) being translated into outputs by some function described only with the box. Activity Diagrams are expressed in terms much closer to our own english language.

The other techniques analyzed for possible use in defining the "As-Is" model of the software development process are Data Flow Diagrams (DFD) and IBM's Hierarchical Input Process Output (HIPO) technique. The DFD technique uses circle, sometimes called bubbles, to define processes or functions. Between these functions are data flows, represented by labeled vectors, that are the inputs and outputs for the functions. Each of these functions may be decomposed into a separate diagram of several other functions to show greater detail.

The IBM HIPO technique uses a hierarchical structure of specification modules to describe system functions. Each specification has three sections, one for inputs, one for outputs, and one for process. Data arrows between these sections show the relationships between the individual components of each section.

Softech's SADT: Structured Analysis and Design Technique (Ref 79) is chosen to describe the "As-Is" model of the software development process. The SADT Activity Diagram methodology is used to describe the software development process because of it's ease of understanding and facilities for defining mechanisms. Mechanisms are important for identifying where automated support is required in the development process. The differentiation between input and control for the data items is also of significance when describing the software development process.

The two other techniques considered for depicting the "As-Is" model of the software development process were the Data Flow Diagram (DFD) technique and IBM's Hierarchical Input Output Process (HIPO) technique. The Data Flow Diagram technique was not chosen because it lacks the capability to illustrate where automation and interactive capabilities need to be applied to the software development process. The HIPO technique was not selected for this model because it fails to allow the depiction of complicated flows of information (data) between processes. Of the three techniques considered, the SADT was the only one that allowed both the complicated data flows to be represented and the application of mechanisms to be specified.

The development of the SADT model of the software development process is fundamental to the understanding of that process. However, the model is large and used primarily as background for the SDW Requirements Definition. For these reasons, the model and the associated textual information are included as Appendix A.

### 2.3 SDW Objectives and Concerns

Prior to developing the functional model of the Software Development Workbench, an extensive literature search is conducted to identify the objectives and concerns. These objectives and concerns must be considered when developing an automated software development environment. Especially within the past two years, a number of renowned computer science researchers have been investigating automated software development environment concepts (Ref 59;60;89;33;67;38). These publications have revealed an extensive list of objectives and concerns that are addressed by the SDW development effort. The objectives of a software development environment depend on the types of activities that the environment is to support. A certain set of objectives are fundamental to all software development environments, while other objectives are more characteristic of specific types of environments. The objectives of the AFIT SDW are common to most software development



environments. However, they are limited to the technical development of software and not the managerial objectives that would be required by an environment supporting larger multiprogrammer development efforts. The objectives of the AFIT SDW are described in the following paragraphs.

2.3.1 The Reduction of Software Errors. A primary objective of an automated software development environment should be the reduction of as many software errors as possible and the early detection of those that occur. There are many types of software errors, but those errors that deal with software reliability are by far of the most significance here. Software errors have accounted for tremendous losses of money, equipment, and, most importantly, human life (Ref 32). A simple software error was responsible for the crash and destruction of an early martian landing vehicle costing billions of dollars (Ref 22). Software errors in aircraft control software could cost the lives of air crew members. In a ballistic missile detection system, software errors could, theoretically, result in the worst of all possible disasters, a nuclear exchange (Ref 12). Within more common software systems, errors have caused costly budget overruns and schedule slippage.

Software errors may be classified in many ways. One manner of distinguishing them is by the stage of the software development in which they are detected (Ref 32). The cost of correcting errors increases exponentially as they are detected later in the development cycle (Ref 57). Many times the action taken to correct for one error creates more errors. The likelihood of this occurring also increases exponentially as the error is detected later in the development cycle (Ref 57).

Another manner of distinguishing errors is by the stage of the development in which they are made. Analysis and design errors which are made early in the development are easily corrected if they are detected early. However, if they are not detected until later in the development effort, they become the most costly errors to correct (Ref 12:2). Implementation errors on the other hand occur late in the development and are usually much less expensive to correct. The early detection of system software errors is thus of extreme importance. In many software projects up to 50% of the budget is spent on maintenance of software errors, while only 17% of the budget is spent on analysis and design (Ref 90:29). The logical response to this problem would be to invest much greater effort on the design and analysis stages of development in order to discover, isolate, and remedy errors early in the development cycle. An emphasis on the design and analysis phases of development, the utilization

of state-of-the-art software engineering methodologies and automated internal and external validation, would assist in detecting errors much closer to when they occur. Thus, the cost and risk associated with fixing them would be reduced.

2.3.2 Responsiveness to Change. Even with a very strong emphasis on the analysis and design stages of development, errors are likely to occur. The user's requirements for the software are also likely to change either during or after the software's development. Modifications to a software system are inevitable and mechanisms to handle them must be built into the SDW. Changes to the system must be well documented, so as not to repeat an error previously made, and all system documentation must be kept consistent as changes occur.

2.3.3 Rapid Assessment of Design Alternatives. During the design of a software system, many design decisions must be made. Often these decisions are made with very little assessment or understanding of alternatives due to time and other constraints. An automated interactive software development environment should provide means of rapidly assessing the consequences of different design alternatives. Contemporary technology offers a variety of simulation tools (Ref 53:334), which if properly used, could produce feedback on design decisions in near real-time.

2.3.4 Automated Documentation Support. The production and maintenance of software system documentation has been and continues to be a major source of frustration for software developers. With current technology, the automated production and maintenance of this documentation could be implemented, thus greatly improving the efficiency of the software development process.

2.3.5 Software Managerial Capabilities. While not a primary objective of the initial development of the SDW, the improvement of software management techniques is of great importance to the Air Force. Therefore, facilities for improving the management of software should become objectives of later versions of the SDW. This could be accomplished by keeping time and manpower statistics on all development efforts, thus allowing for the better scheduling and resource planning of future development efforts. Facilities that estimate the status of a current development effort would also be of great benefit to the software manager.

The objectives of any development effort are of primary importance and the SDW development effort is no exception. However, the manner in which the objectives are accomplished must be guided by a set of concerns that are also fundamental to the proposed system. The concerns that guide the development of the SDW are a summarization of those

discussed by leading authorities on the subject of software development environments. The varied opinions of these authorities provide quite a list of concerns fundamental to the development of an environment. However, many of these concerns overlap or are not of significance to the SDW development effort. Thus, the list and description of concerns that follows is a synthesis of the authorities' opinions and the needs of the AFIT software community:

- 1- Integration
- 2- Traceability
- 3- User-Friendliness
- 4- Testability
- 5- Pre-Fabricated Programming
- 6- Support the Entire Software Life-Cycle
- 7- Flexibility
- 8- Consistency and Completeness
- 9- Explicitness and Understandability
- 10- Documentation Support
- 11- Updateability
- 12- Language Independence
- 13- Early Prototyping

2.3.6 Integration. The issue of integration is considered to be a primary concern of software development environments that must support the development of software throughout the entire life-cycle. Automated tools exist to support every stage of the software life-cycle. Most of these tools provide very effective means for accomplishing the objectives of the stage they support. The fundamental problem has been that these tools are not compatible with the tools that support the other stages of development (Ref 89:8). Achieving a fully integrated environment is a very difficult task. However, integration can be realized at many levels of detail and is thus not only a concern of the initial SDW development but also a concern that will drive the evolution of the SDW in the future.

2.3.7 Traceability. The SDW must support all stages of software development and to do so the developer must be able to trace between the development stages. This ability to trace between development stages is fundamental to the validation of the later stages products against the results of the previous stages (Ref 89:8). Both forward and backward tracing between development stages should be supported.

2.3.8 User-Friendliness. User-friendliness is a term often used in computer science circles. Difficulty of use for both the experienced and the un-experienced user has

constrained many users from fully realizing the computer's potential. Within the SDW project, user-friendliness can be analyzed as two separate concerns. The teachability of the environment and the human factors engineering of the environment are the important aspects of the broad concern of user-friendliness. The emphasis of both of these concerns should be on tools whose user interface emphasizes the function of the tool and not how to get the tool to perform its function (Ref 33:46).

Teachability refers to the level of ease with which an unexperienced user can become comfortable with and useful on the system. The primary users of the SDW are to be AFIT students and professors. Neither of these groups of individuals have very much time to learn how to use a new system. The SDW must be easily learnable and possess means to walk users through its operation.

Human factors engineering is one of the most dynamic areas within the computer science discipline (Ref 33:52-53). New and innovative manners of interfacing with the computer are being developed at an alarming rate. The resulting new technologies should be used within the SDW to the maximum extent possible. Interfacing with the SDW should be able to be accomplished with a variety of means. Easy to learn and use command languages should be used for both environmental and tool interfaces. Extraordinary work has been done with

Artificial Intelligence in recognizing requests given in natural language forms (Ref 8). Interactive graphics capabilities should be utilized for working with the graphical software engineering methodologies supported by the SDW. Advances in Pattern Recognition and Speech Synthesis could allow the user to interface with the SDW by means of a verbal conversation (Ref 89).

2.3.9 Testability. Testing and validation are often tedious and time consuming operations. However, the internal and external validation of the products of each of the development stages is fundamental to successful software development. Capabilities for automated and interactive testing should be available to validate the intermediate and final products of all of the development stages.

2.3.10 Pre-Fabricated Programming. By the time that the Preliminary Design has been established, many of the functional modules, whose algorithms have not yet been developed, may be satisfied with already written modules (Ref 80). Studies have shown that, for business applications, as much 40 to 60 percent of the required modules already existed (Ref 50). The SDW should provide means to utilize existing algorithms and codes within development efforts. Such a facility could significantly reduce development time for many software systems.



2.3.11 Support of the Entire Life-Cycle. This concern has already been stated, but is important enough to re-emphasize exclusively. Automated and interactive tools should be present to support each and every phase of the software development from conception to termination. Furthermore, these tools should be interfaced in manners that will allow them to properly support the entire life-cycle.

2.3.12 Flexibility. The concern of flexibility within the context of an automated software development environment takes on a variety of dimensions (Ref 89). The environment must support many types of software developments including mathematical or scientific applications, real-time and control applications, and data base developments to name a few. Each of these application types requires some set of specialized tools. The environment must be able to support development projects of different sizes. Facilities for handling more than one version of a system must be present. The environment must support several programming languages in order facilitate the choice of a language that is best suited for the particular application. Each language requires specific compilers, debuggers, and other language specific tools. Some languages require additional testing tools to perform testing that is done by compilers in other languages. One example of this is the DAVE tool that checks FORTRAN programs for data anomalies that are automatically

check for in Pascal programs by the compiler (Ref 38). The tools that compose the environment must be tailorable to fit the needs of users with different experience and skill levels (Ref 33).

2.3.13 Consistency and Completeness. Automated tools must provide for the automatic checking for errors, omissions, ambiguities, and redundancies (Ref 2). Facilities are needed for checking the consistency of data names, data types, and data units. Checking of module interfaces for consistency could also be automated. Items such as the number of parameters, their types, and their order would be of importance here (Ref 13). Consistency should be automatically maintained between stages of the development life-cycle. By using the shared data base with a common data format, only one copy of the data exists, instead of separate copies for each of the tools. This elimination of redundancy is imperative in maintaining consistency.

Completeness testing is also fundamentally important during software development. Automated facilities could check for modules that have been referenced but not specified, or insure that all previous stage components have been satisfied by later stage components. Completeness checking is also important from a managerial point of view in order to assess the status of the project at any

particular time.

2.3.14 Explicitness and Understandability. Products of each stage should be as explicit as possible in order to avoid misinterpretations (Ref 2). The products should also be easily understandable to those with limited background with the system. This makes the concern similar to a double-edged sword. Often the rigor of explicitness severely limits the general understandability. For example, mathematical symbology is very explicit and precise, yet it lacks an understandability to the mass of society. In addition to explicitness for the human reader, the product must be explicit to the computer which requires perhaps even a more rigorous format (Ref 83).

2.3.15 Documentation Support. Documentation support is a primary concern of any software development environment because software only exists in its documentation. A major reason that the tools of software engineering are not fully utilized is that they often require a significant amount of additional documentation. This documentation must be generated using development resources, namely personnel. Automation of the documentation involved would allow its production to be less costly and thus more generally utilized. Automated documentation support should use data from the data base to produce both hard and soft copy documentation upon demand (Ref 59). By using the data

directly from the data base the problems of saving and updating the documentation are automatically taken care of.

The environment should have capacities to produce an extensive variety of documentation (Ref 89). Both hard and soft graphics capabilities should be present in the environment. Hard graphics refers to printed graphical illustrations, while soft graphics are illustrations presented on a CRT device that are easily altered. In the past, documentation has been the weak link in system development (Ref 83). The environment's documentation facilities should include mechanisms for developing scripts for a variety of activities (Ref 89) and for the continuing development of user's manuals.

2.3.16 Updateability. The development of most software systems is an iterative process (Ref 83). There is often needs to update previous development data in a precise and consistent manner (Ref 33). Besides just recording the modifications that were made, it is important to record why they were made to avoid repeating the error (Ref 83). Often a copy of the previous version of the module comes in handy if the correction proves to be less optimal than the original. The development environment should support the modification and justification of development data in a simple and consistent manner.

2.3.17 Language Independence. Many programming languages are presently available to the software developer. Each of these languages has its own set of characteristics and features that make it desirable for a particular set of applications. In most software development scenarios, the selection of a programming language is not required until the actual Implementation stage or, at the very least, the Detailed Design stage. Requirements Definition and Preliminary Design should be accomplished prior to selection of the language and the tools used for these stages be independent of any particular programming language. Thus, the selection of the language can be made only after a sound understanding of the system being developed has been established (Ref 33). Language independence can be realized in the later development stages also with generic tools that require only a description of the language constructs.

2.3.18 Early Prototyping. The use of proposed system prototyping has proven a major break through in developing systems that can accurately meet the needs of the user and thus gain full acceptance upon completion (Ref 24). Prototyping of the user interface allows the user to get a feel for the system very early in the develop effort. Functional prototyping insures that the system being developed is what is needed to satisfy the needs of the user. However, the user often does not fully understand what his needs are. A prototype allows the user to gain

familiarity with the proposed system. Thus, the experiences of the user actually drive the design of the system. Rapid and early prototyping is useful in validating the requirements and preliminary designs of the proposed system by illustrating inadequacies and discrepancies (Ref 74).

2.3.19 AFIT Specific Objectives and Concerns. The objectives and concerns of the SDW development are very general and characteristic of any good software development environment. However, the SDW development is specifically geared for use by the AFIT software community. Specific needs and requirements exist for this community, many of which fall into the previously stated categories, but some that do not. A description of the requirements of the SDW that are specific to the AFIT software community is included to help insure that the SDW is developed in a manner that is of benefit to this community.

The SDW is intended for use by two categories of users within the AFIT software community. First, the SDW is to be used by students enrolled in the AFIT Software Engineering course (EE 5.93). The SDW is also to be used by students and faculty involved in thesis and other extensive software development efforts. Each of these two categories of users have particular requirements for the SDW development. The students enrolled in the Software Engineering course are to use the SDW as a pedagogical tool to learn and gain

experience with the classical Software Engineering methodologies. Examples of these Software Engineering methodologies would include Data Flow Diagrams, HIPO charts, SADT(tm) diagrams, Data Structure Diagrams, Structure Charts, Structured English, etc... The interactive tools within the SDW that support these methodologies should be easy to learn and to demonstrate. Furthermore, both the SDW executive and the component tools should provide on-line training facilities. The component tools should stress the principles of the supported methodologies and not the operations of the tools.

The other category of SDW users is the students involved in thesis research and the students and faculty involved in other extensive software development efforts. A variety of many different types of software developments take place with the AFIT software community. The SDW must provide means to support the full range of these development efforts. Examples of the types of software development efforts currently underway include efforts to develop relational data base management systems, to develop interactive graphics languages, to implement concurrent and distributed software systems, and to develop numerical calculation software for the study of control systems. As the SDW development progresses in follow-on thesis efforts, the SDW could be extended to a systems development environment, that supports the hardware, software, data base

and other components of a system's development. For the near term, the SDW must provide for a number of distinct, yet concurrent, developments to be supported with proper security and separation of development data. Since many of the software development efforts within AFIT are part of a continuing series of developments on a single software system, the capability to archive the development data for future use must be provided. The documentation produced by the SDW should be of a very high quality to allow for its inclusion into the formal reports and thesis manuscripts. Additionally, the SDW and the component tools must be easy to learn and use to allow the SDW users to concentrate on their particular development effort and not on the operation of the SDW.

The primary difference between the SDW and other software development environments is that the SDW does not require capabilities to assist the managers of software development efforts. The development activities within the AFIT software community are limited to one or two person efforts, thus the need for extensive managerial capabilities is not present.



#### 2.4 Functional Model of the Software Development Workbench

With the "As-Is" model of the software life-cycle provided by Appendix A and the preceding list of SDW objectives and concerns, a sufficient background has been established for the development of a definition of the SDW functional requirements. A variety of methodologies exist for defining and describing requirements. From this variety, the Data Flow Diagram technique is used to define the requirements for the SDW. Data Flow Diagrams, often referred to as "Bubble Charts", illustrate operations on data items by circles or bubbles and flows of data between the operations by labeled vectors as shown in Figure 3.

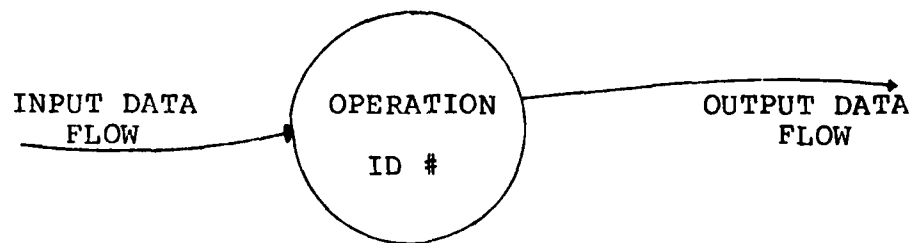


Figure 3: Data Flow Diagram Constructs

A data flow is an abstract data item that moves from a source to a destination. Sources may be inputs to the system or outputs of operations that produce the data item. Destinations may be outputs for the system or inputs to operations that alter the data item. Each Data Flow Diagram may contain several operations and data flows. Operations

on a diagram may be broken down into separate diagrams, thus forming a hierarchical structure of increasing detail (Ref 89).

Data Flow Diagrams are especially useful in describing requirements for systems which specify a complex array of data flows. The SDW is this type of system. Mechanisms, or the "hows" behind the operations, are left out of the Data Flow Diagrams, as they should be when specifying requirements. Data Flow Diagrams are easily translated into Preliminary Design Structure Charts by using techniques such as Transaction Analysis and Transform Analysis (Ref 89). Data Flow Diagrams provide for flexibility by allowing multiple levels of detail to be illustrated. For these reasons the Data Flow technique is used to specify the requirements of the SDW.

The Data Flow diagrams used to describe the requirements for the SDW are explained in textual supplements. Each textual supplement makes references to the data flows and operations of the corresponding diagram. The titles of these data flows and operations are capitalized to allow the reader to identify them more easily.

The functional model for the SDW utilizes DFDs in a hierarchical structure that extends down four levels. Each DFD has an accompanying textual description to aid the reader in understanding the diagram. Both the data flows and the operations are capitalized when mentioned within the textual supplements. The data flow in each diagram are described in the SDW Data Dictionary that is included as Appendix B. The diagram outline below (Figure 4) is provided to assist the reader in understanding the breakdown of the model.

### SDW Functional Model Outline

Figure Number	Diagram Title
5	0- SDW Functional Model: Top Level
6	1- Perform Software Life-Cycle
7	1.1- Perform Requirements Definition
8	1.1.1- Develop Draft Requirements
9	1.1.2- Translate Requirements into Machine-Readable Form
10	1.2- Develop Preliminary Design
11	1.2.1- Develop a Draft Preliminary Design
12	1.2.2- Validate Preliminary Design
13	1.3- Develop Detailed Design
14	1.4- Implement and Test the Software System
15	1.4.3- Convert to Syntactically Correct Code
16	1.4.4- Test Code with Traces and Error Handling
17	1.4.5- Optimize the Code
18	1.5- Integrate to and Validate on Target Machine
19	1.7- Maintain and Operate Software System

Figure 4: SDW Functional Model Outline

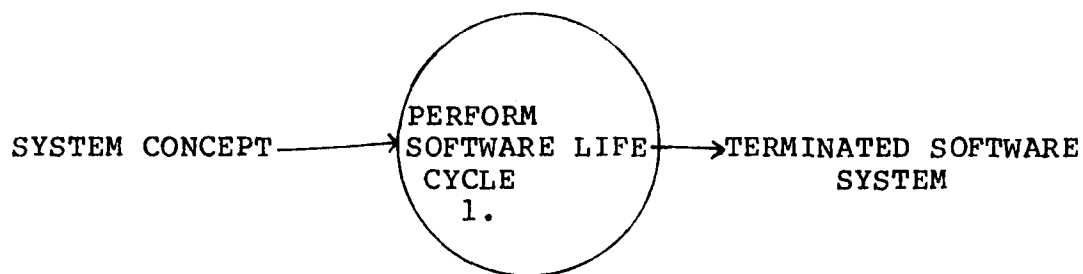
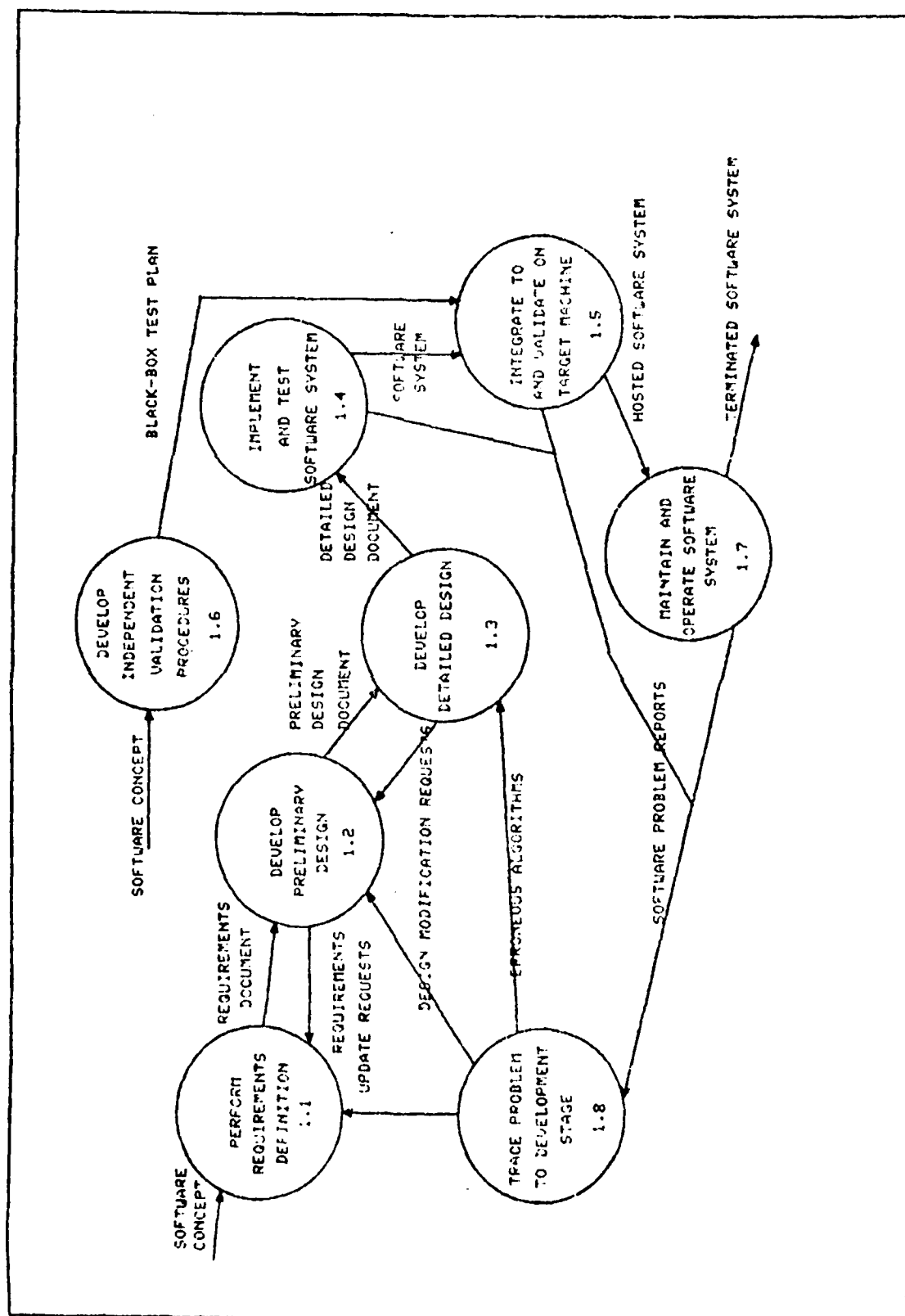


Figure 5: SDW Functional Model: Top Level

2.4.1 The SDW Functional Model: Top Level. This top level diagram of the SDW requirements illustrates the scope of the SDW investigation. The System Concept is a vague and ambiguous idea for a software system. The Perform Software Life-Cycle operation is the process of developing, operating, and maintaining the software system. The requirements within this operation are what the SDW must satisfy. The SDW must support this Perform Software Life-Cycle Operation until the software system is no longer needed, at which pointed it is terminated.



2.4.2 Perform Software Life-Cycle. This level of the SDW Functional Model is a breakdown of the Perform Software Life-Cycle operation of Figure 5. The breakdown is accomplished by dissecting the life-cycle into its component stages. The six stages of the life-cycle are represented as operations with their input and output data flows shown accordingly. Perform the Requirements Definition is the first stage (operation 1.1). This operation translates the vague System Concept into a detailed Requirements Document. Requirements Update Requests are also inputs to the Perform Requirements Definition operation. This illustrates the possibility that experience gained later in the development cycle may require the existing Requirements Document to be updated.

Developing the Preliminary Design is the next operation (operation 1.2). Inputs to this operation are the Requirements Document and Design Modification Requests, which again demonstrate the requirement for modifiability. Likewise, the Develop Detailed Design operation (operation 1.3) translates the Preliminary Design Document into a Detailed Design Document and accepts reports of Erroneous Algorithms as input. These reports require a modification to the Detailed Design Document. Both the Preliminary and Detailed Design operations may output Requirements Update Requests if the present Requirements Document is not sufficient.

The Detailed Design Document is translated into a Hosted Software System by a sequence of two operations (operations 1.4, 1.5) that implement, test, integrate, and validate the software system. Software Problem Reports may be generated by these operations if the previous development products prove to be insufficient. The Integrate to and Validate on Target Machine operation (operation 1.5) accepts a Black-Box Test Plan. This test plan is utilized for independent validation. The Black-Box Test Plan is a product of the Develop Independent Validation Procedures operation (operation 1.6) that occurs concurrently and independently of the rest of the system development. The Hosted Software System is accepted by the Maintain and Operate Software System operation (operation 1.7). The output of this operation is either a Terminated Software System or a Software Problem Report. The Software Problem Reports are feed to the Trace Problem to Development Stage operation (operation 1.8). This operations decides at which stage the problem should be remedied. The inclusion of this operation in the SDW Functional Model illustrates a major difference between the existing software life-cycle shown in Appendix A and the life-cycle supported by the SDW. The existing life-cycle emphasizes the idea of "quick fixes" to the software, whereas the life-cycle supported by the SDW promotes the re-development of the software from the point of error occurrence. The approach taken by the SDW greatly



reduces the chances for the introduction of new errors and improves the overall reliability of the software.

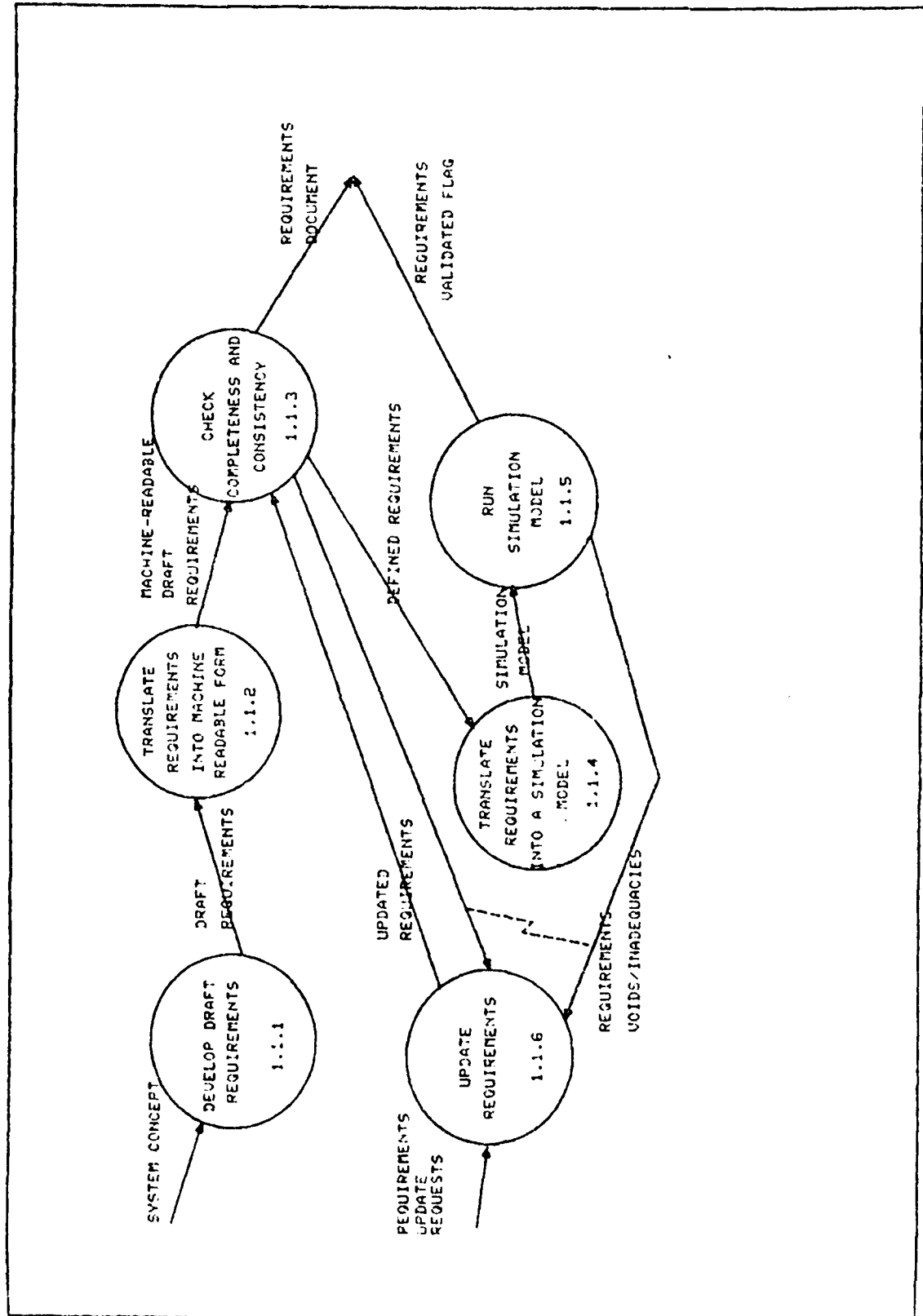


Figure 7: Perform Requirements Definition

2.4.3 Perform Requirements Definition. The Perform Requirements Definition operation specifies that the System Concept be used to develop the Draft Requirements (operation 1.1.1). The resulting Draft Requirements are translated into a machine-processible form (operation 1.1.2). These Machine-Readable Draft Requirements are checked for consistency and completeness (operation 1.1.3). If Requirements Voids/Inadequacies are discovered, the Draft Requirements are updated (operation 1.6). Requirements Update Requests are also processed by operation 1.1.6. The Updated Requirements are then be check for consistency and completeness again (operation 1.1.3). If the stated requirements are consistent and complete, they are defined as the Requirements Document. Before this document is passed on to the next stage, it may be translated into a Simulation Model for further internal validation (operation 1.1.4). Running of this Simulation Model (operation 1.1.5) may reveal Requirements Voids/Inadequacies. These Voids/Inadequacies are processed by the Update Requirements operation and the Requirements Document is re-compiled.

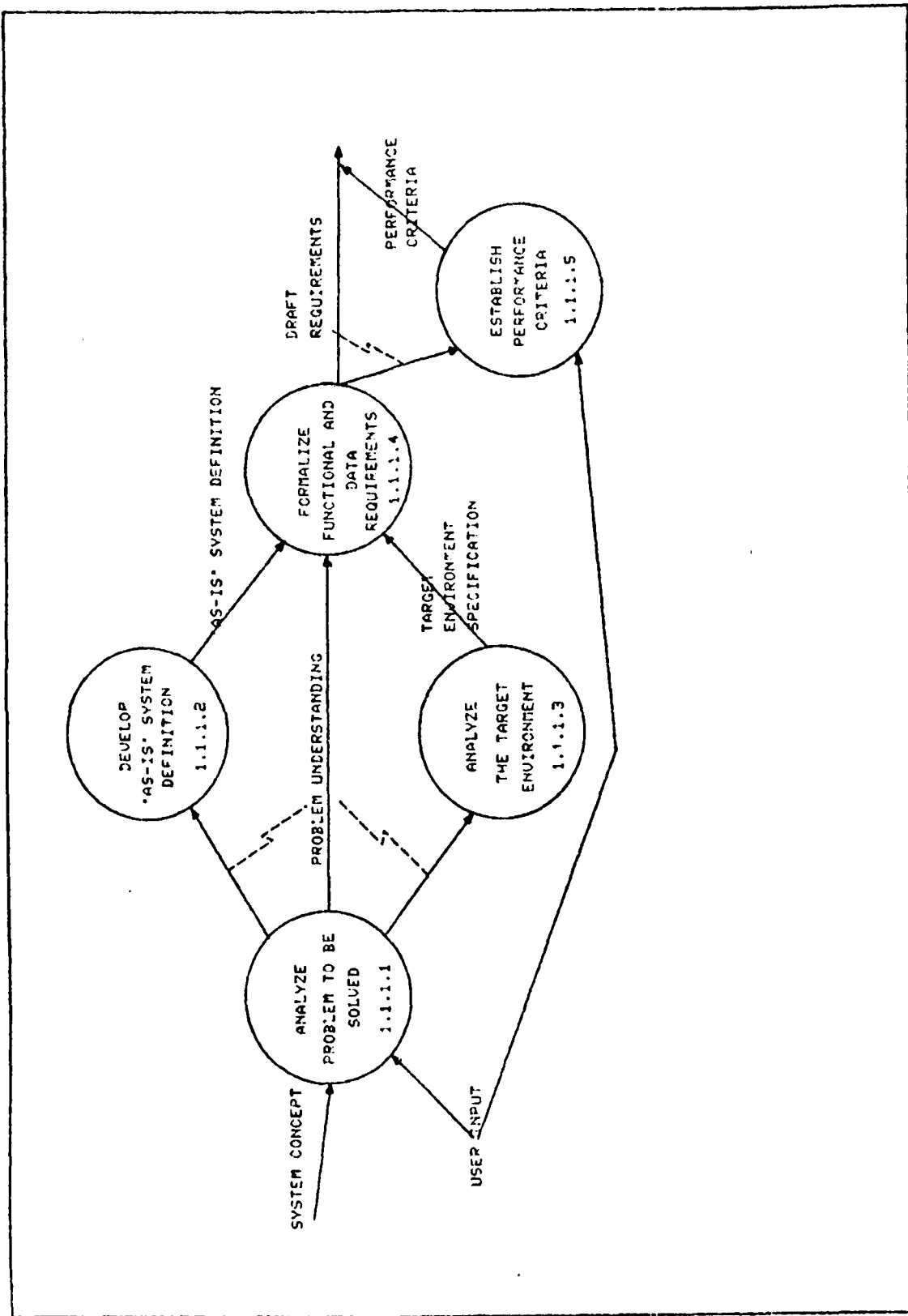


Figure 8: Develop Draft Requirements

2.4.4 Develop Draft Requirements. Developing the Draft Requirements is a very important function that requires a knowledge of the System Concept and, possibly, User Input. The first step (operation 1.1.1.1) is to carefully analyze the System Concept and User Input to gain a Problem Understanding. With this Problem Understanding an "As-Is" System Definition may be developed (operation 1.1.1.2), similar to Appendix A of this thesis investigation. An analysis of the Target Environment (operation 1.1.1.3) may also be necessary if the Target Environment possess uncommon attributes. The Problem Understanding, "As-Is" System Definition, and Target Environment Specification are all used to Formulate the Functional and Data Requirements for the software (operation 1.1.1.4). The result is a set of Draft Requirements that may be enhanced with a set of Performance Criteria (from operation 1.1.1.5) if the particular system requires.

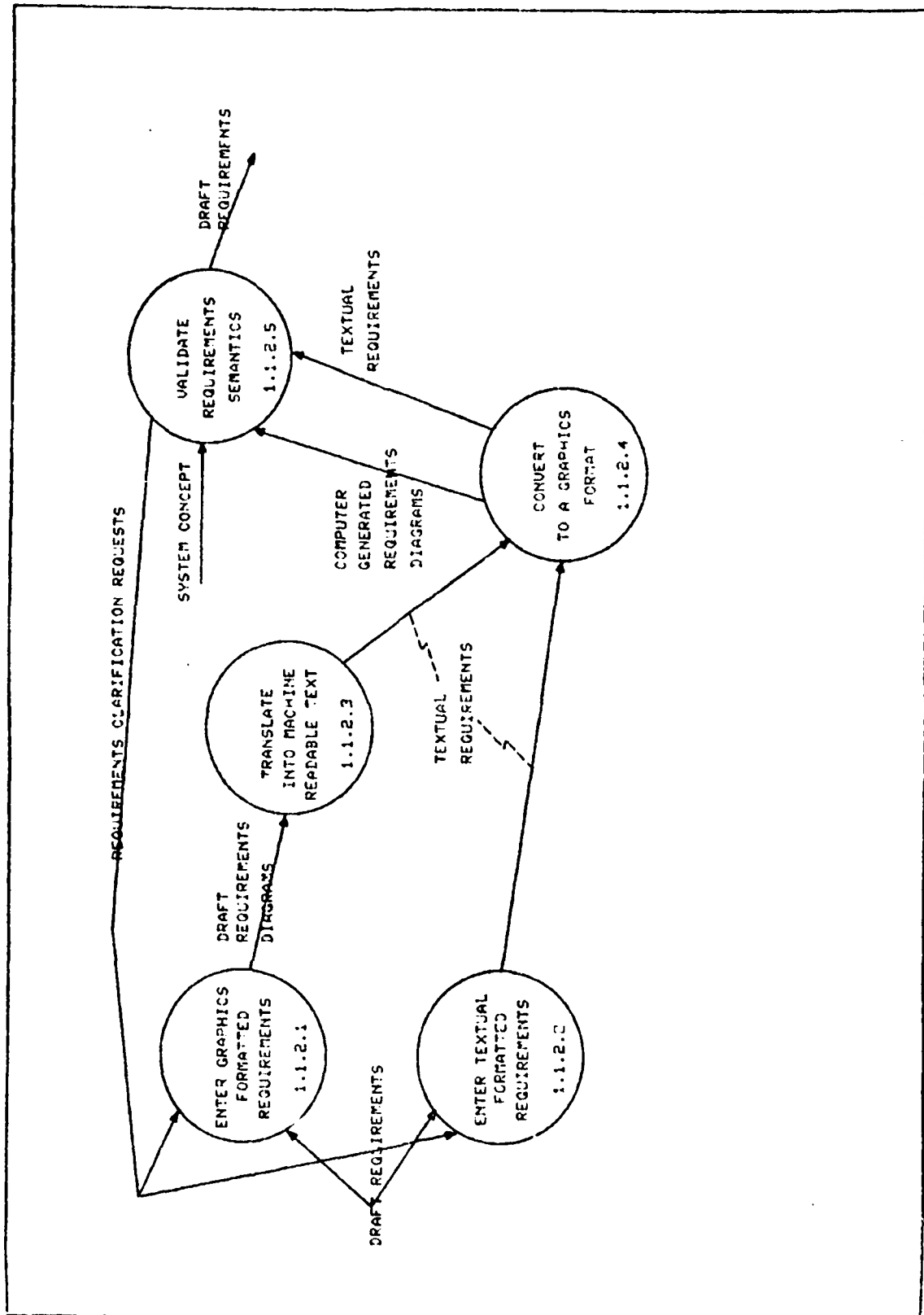


Figure 9: Translate Requirements into a Machine-Readable Form

2.4.5 Translate Draft Requirements into a Machine-Readable Form. The Draft Requirements could be entered using either graphic or textual mediums. Most Requirements Definition tools only accept textual inputs and thus textual input must be supported (operation 1.1.2.2). However, since most Requirements Definition methodologies utilize graphics for understandability, the ability to enter the graphical representations directly (operation 1.1.2.1) greatly simplifies the process. If graphics are used, they must be translated to a textual format for computer processing (operation 1.1.2.3). The translation of the textual statement of the requirements into a graphic form (operation 1.1.2.4) allows the easier validation of the exact stated requirements against the user's or developers perception of the System Concept (operation 1.1.2.5). This process is often referred to as verification within the computer science community. Generically stated in terms of the computer science community, verification is simply the checking that what the designer/programmer has told the computer to do is what he meant to tell the computer to do.

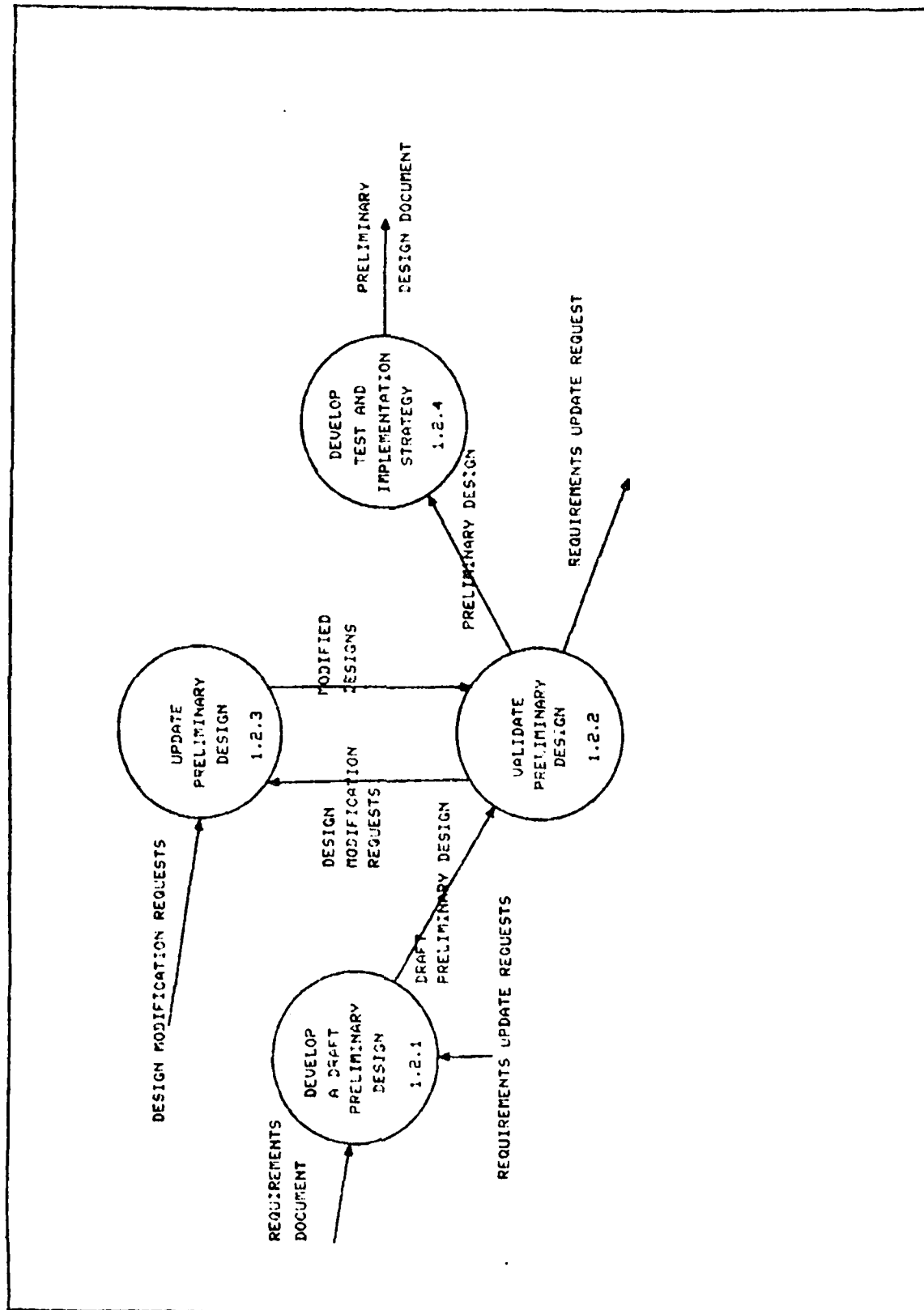


Figure 10: Develop Preliminary Design



2.4.6 Develop Preliminary Design. The developing of a Draft Preliminary Design could be accomplished by using any one of a number of design techniques (operation 1.2.1). Ideally, several different design techniques are supported by the SDW to allow the SDW user to choose the technique best fitted to his individual development effort. The Draft Preliminary Designs are validated against the previously stated Requirements Document (operation 1.2.2). This operation is elaborated on in the next diagram. The output of operation 1.2.2 is either Design Modification Requests, Requirement Update Requests, or a statement of the Preliminary Design. Design Modification Requests is routed to the Update Preliminary Design operation (operations 1.2.3). Updating of the design requires re-validation of the Modified Designs. Once a statement of the Preliminary Design has been developed, it is fed into the Develop Test and Implementation Strategy operation (operation 1.2.4). The Test and Implementation Strategy developed by this operation is included as part of the Preliminary Design Document. This strategy is an incremental plan for implementing and testing of the software system according to a hierarchical design of increasing detail (Ref 89).

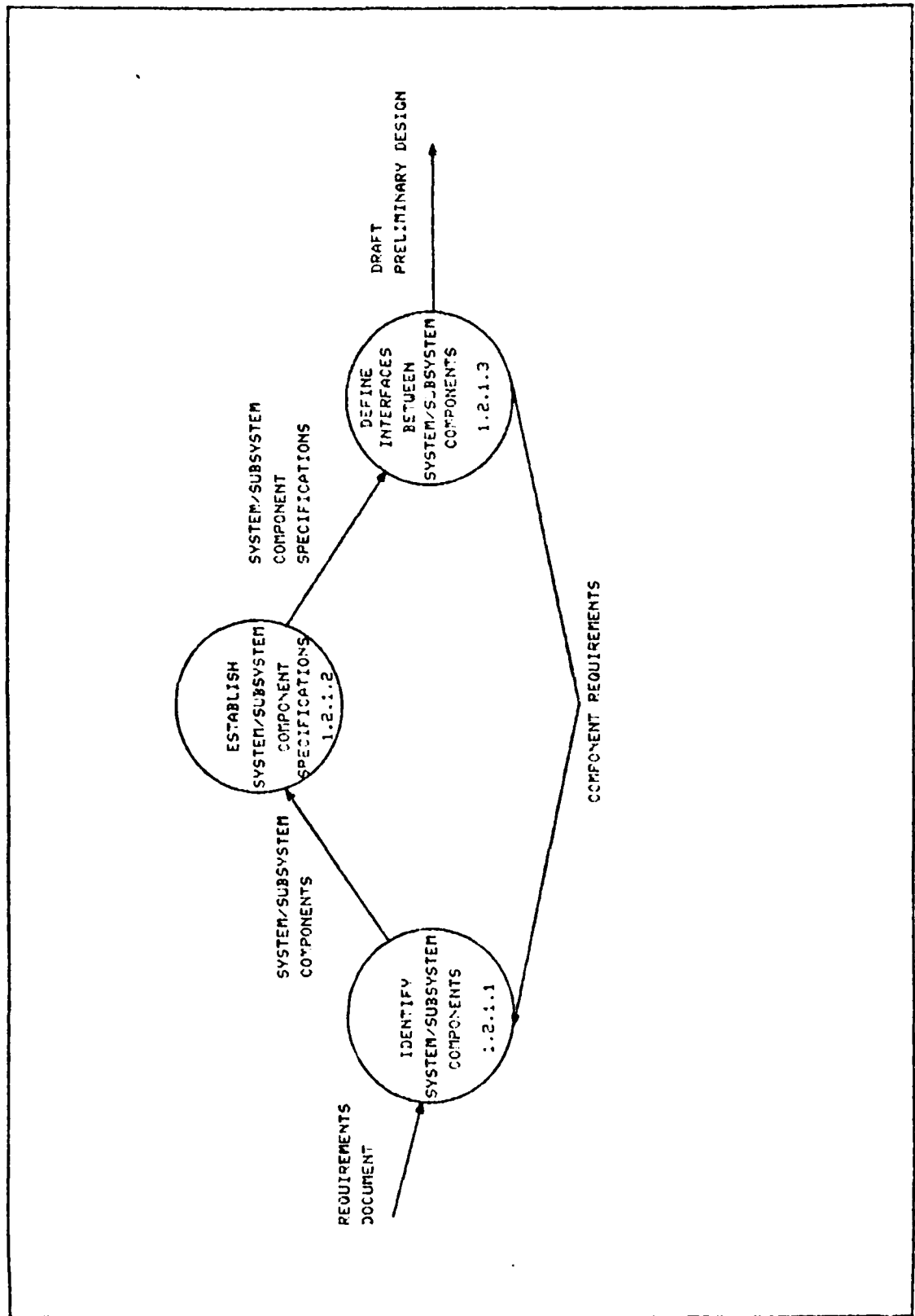


Figure 11: Develop a Draft Preliminary Design

2.4.7 Develop a Draft Preliminary Design. The Development of a Draft Preliminary Design is an iterative process. The Requirements Document specifies the software system to be designed. The System/Sub-System Components are then identified (operation 1.2.1.1). The System/ Sub-System Components Specification are then established (operation 1.2.1.2) and the interfaces between the components are defined (operation 1.2.1.3). The System/Sub-System Component Specifications that are not down to a single function level are broken down into further Sub-System Components and the process is continued. Once all of the System/Sub-System Component Specifications are down to just a single function, the Draft Preliminary Design is completed.

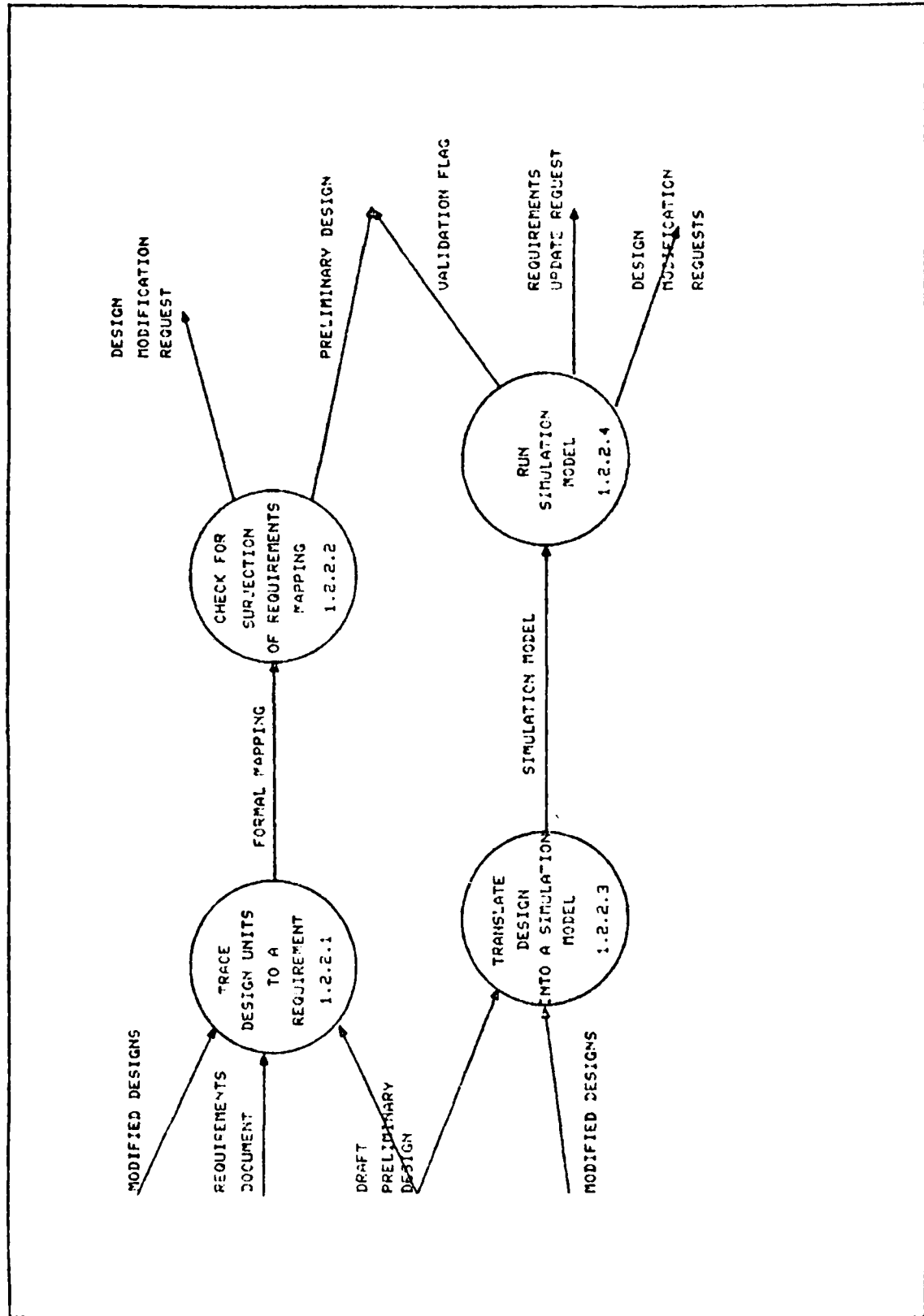


Figure 12: Validate a Preliminary Design

2.4.8 Validate Preliminary Design. Validation of the Preliminary Designs must be done both internally, checking for the completeness of the design, and externally, comparing the designs to the statement of requirements. Internal checking is facilitated by translating the preliminary design into a simulation model and running the model (shown in operation 1.2.2.3 and 1.2.2.4). External validation against the stated system requirements involves the tracing of the design units back to a specific requirement and then checking that all of the stated requirements have been satisfied by at least one design unit (operations 1.2.2.1 and 1.2.2.2). The tracing of design units to requirements should be done in both directions to facilitate later bi-directional movement between the stages.

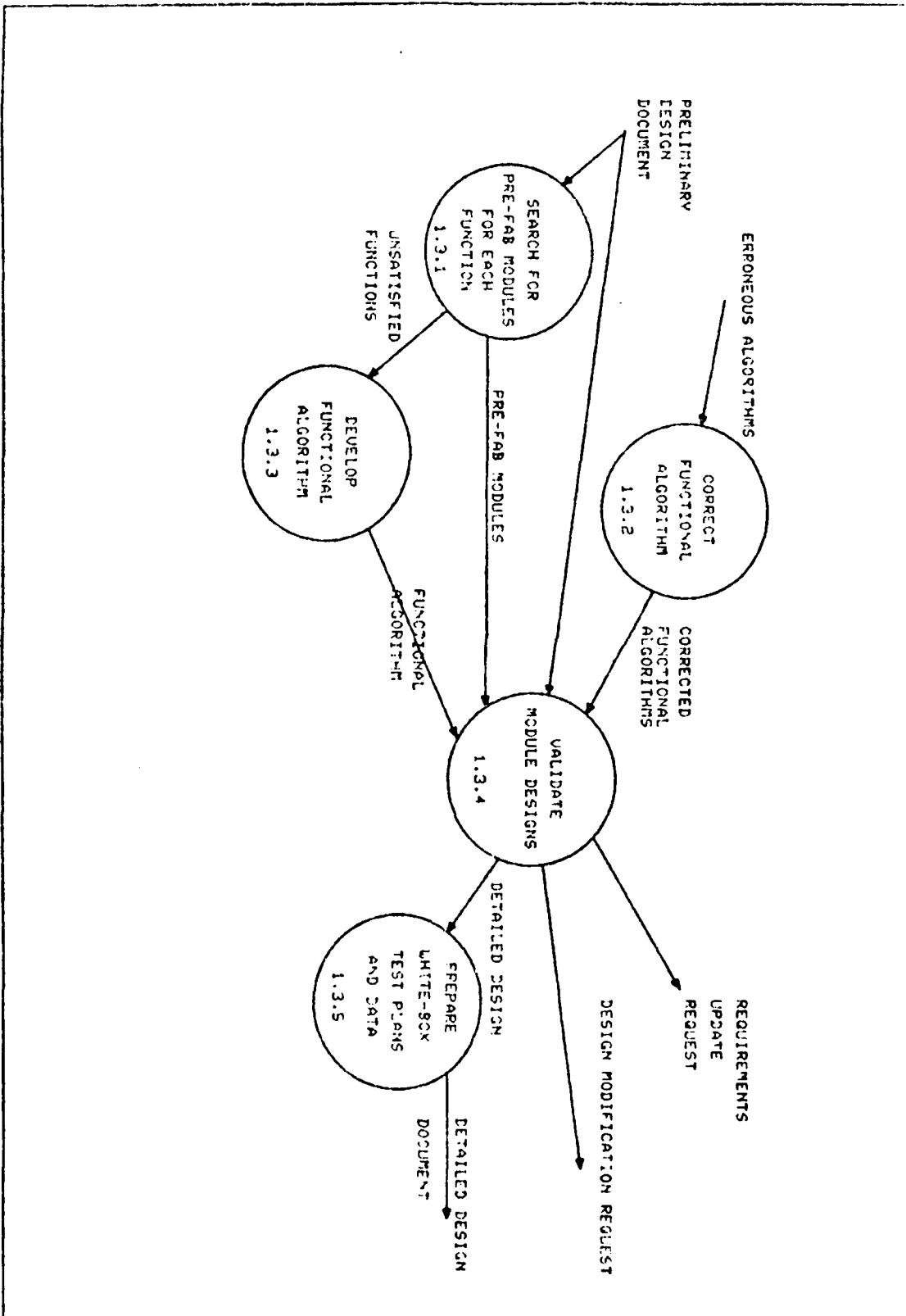


Figure 13: Develop Detailed Design

2.4.9 Develop Detailed Design. As defined in the first chapter, the Detailed Design stage deals with the development of functional algorithms for each of the system modules. Many times, these algorithms may already have been developed and perhaps even coded. Operation 1.3.1 searches for the existence of already developed and coded algorithms to satisfy the requirements of system modules. Those system modules left unsatisfied must be developed by the programmer/software developer as shown by operation 1.3.3. Acknowledging the iterative nature of the software life-cycle, there may exist erroneous algorithms that require correcting (operation 1.3.2). Once the Module Designs have been established, they are validated against the system's requirements and checked for consistency both internally and with the Preliminary Design (operation 1.3.4). If inadequacies are discovered, they are passed to the proper stage in the form of a Requirement Update Request or a Design Modification Request. Otherwise, the specifications of the Detailed Design are used to enhance the Test and Implementation Strategy to include meaningful sets of test data and detailed modular test plans that execute all of the logical paths of the modules (operation 1.3.5).

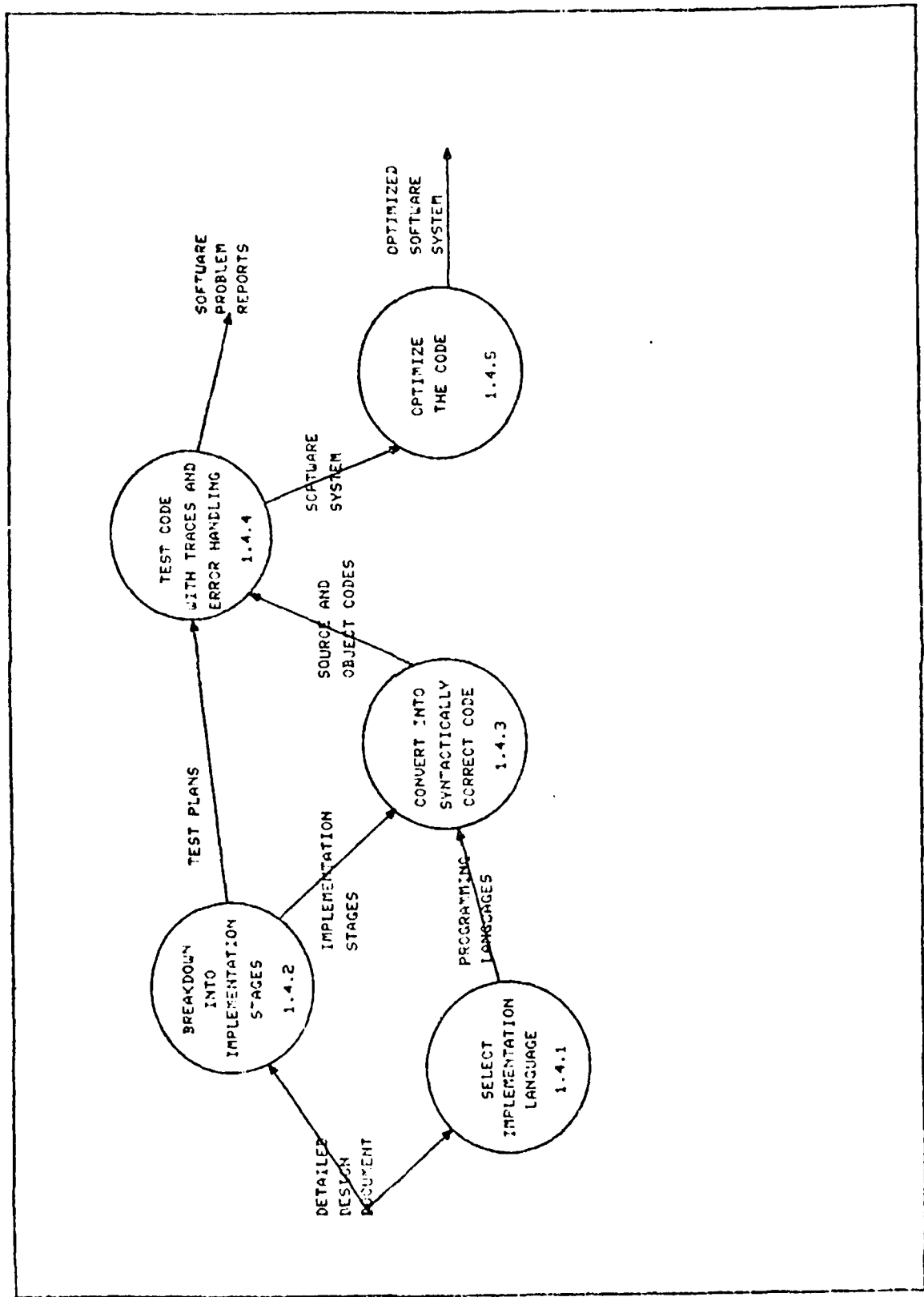


Figure 14: Implement and Test Software System



2.4.10 Implement and Test Software System. Inorder to implement the software system, a suitable programming language must be chosen (operation 1.4.1). Many programming language exist today and each one possess features that make it attractive for use in solving a particular class of programming problem. Careful selection of a programming language can significantly simplify the task of coding. The SDW must support as many languages as possible to provide the software developer the needed flexibility to code his software efficiently. The SDW should possess facilities for only allowing syntactically correct code to be entered (operation 1.4.3). Other facilities of the SDW may provide for some automatic coding by using the products of the Detailed Design stage.

The Test and Implementation Strategy is then broken down into the designated stages (operation 1.4.2) and the objectives of each stage are used to direct the coding operation (operation 1.4.3). The coded section of the system is then tested according to the plans recorded in the Implementation and Test Strategy (operation 1.4.4). After the entire system has been coded and tested according to the Implementation and Test Strategy, the coded system may be optimized as required by an stated performance requirements (operation 1.4.5).

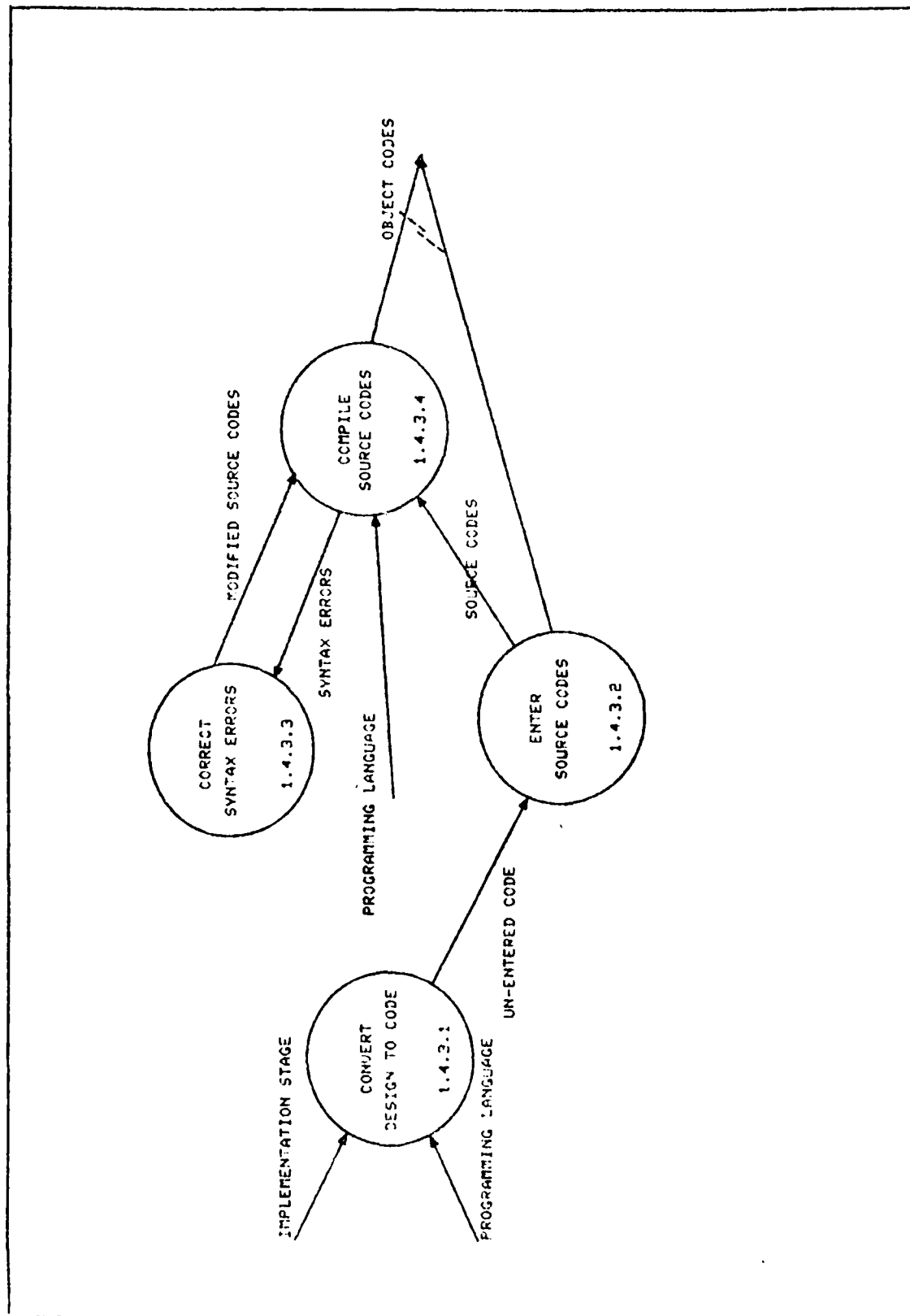


Figure 15: Convert to Syntactically Correct Code

2.4.11 Convert to Syntactically Correct Code. The first step in the conversion of the Detailed Design into code is to convert the Implementation Stages into an Un-Entered version of Code in the selected Programming Language (operation 1.4.3.1). The Un-Entered Code is then entered into the host computer (operation 1.4.3.2) and the result is the Source Code for that Implementation Stage. The Source Code is then compiled (operation 1.4.3.4) which produces the executable Object Codes and detects any Syntax Errors. If Syntax Errors are detected they are corrected (operation 1.4.3.3) and then the Modified Source Code is re-compiled. The result of this entire process is a set of Object and Source Codes for the particular Implementation Stage.

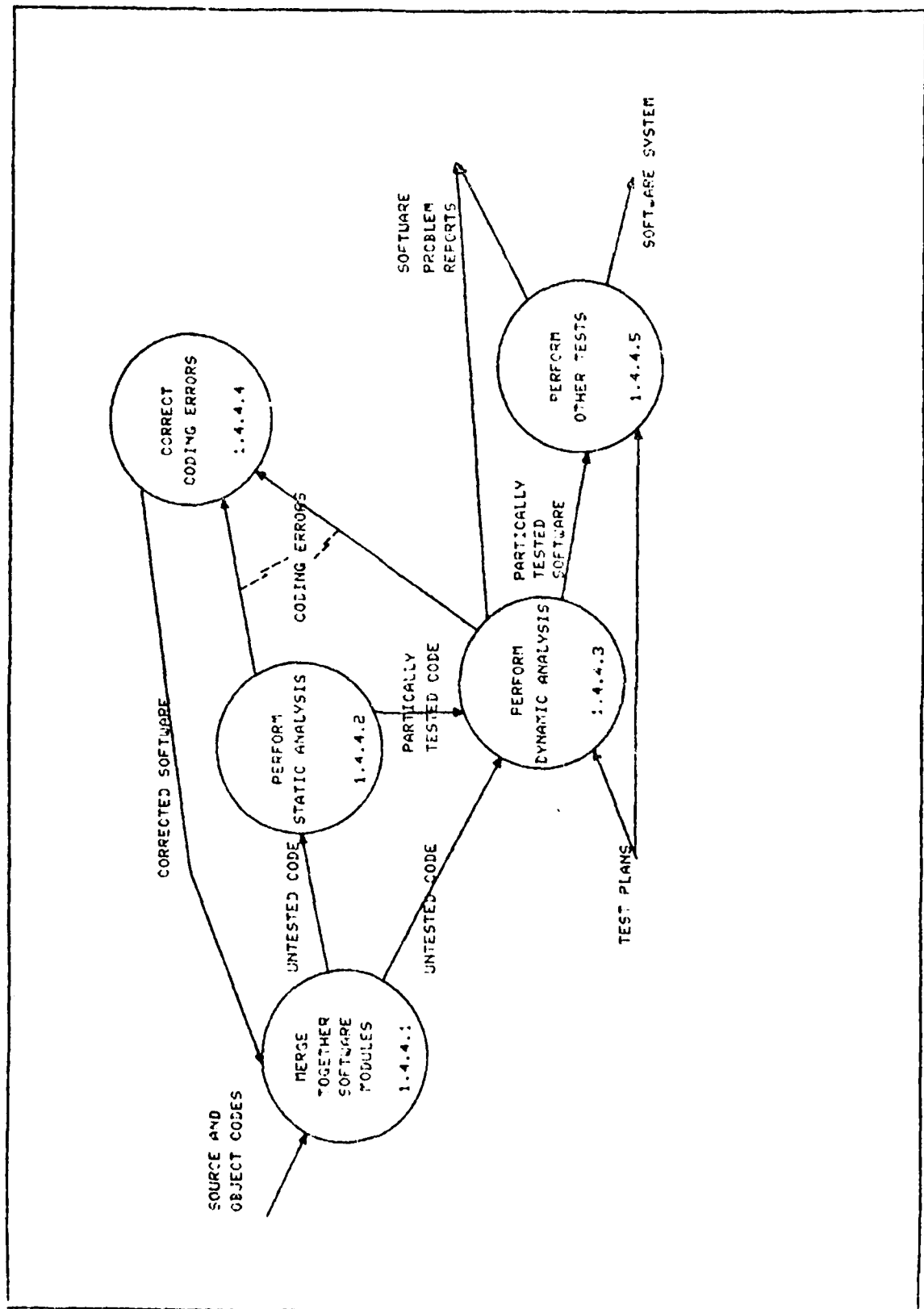


Figure 16: Test Code with Traces and Error Handling

2.4.12 Test Code with Traces and Error Handling. The first step in the testing of the code for each Implementation Stage is to merge together the necessary object codes (operation 1.4.4.1). The resulting Un-Tested Software is then tested using either just Dynamic Analysis techniques (operation 1.4.4.3) or Static Analysis techniques (operation 1.4.4.2) together with the Dynamic Analysis. If Coding Errors are detected, they are corrected (by operation 1.4.4.4) and then re-linked and re-tested. Other tests may also need to be run in accordance with the Test Plans (operation 1.4.4.5). A Software System results once all tests have been passed. If an error is detected that is from a source other than the coding activity, a Software Problem Report is issued.

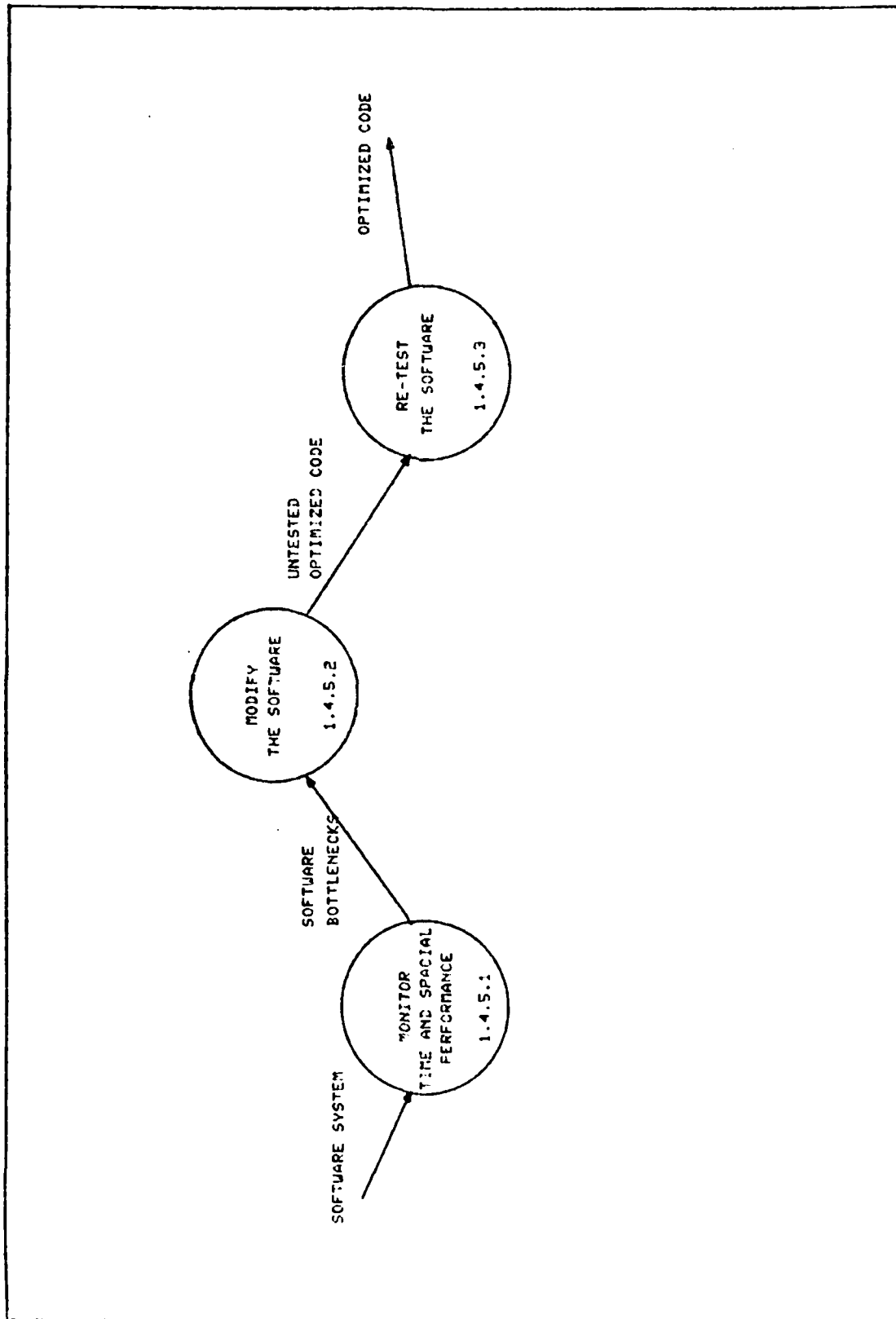


Figure 17: Optimize the Code

2.4.13 Optimize the Code. Often the coded Software System does not execute within the Performance Criteria established for it. If this is the case, the code must be optimized. The first step is to Monitor the Time and Spacial Performance of the Software System (operation 1.4.5.1). This identifies the Software Bottlenecks, which are the areas of the code where the greatest gains in performance can be realized. These areas of the code are then modified (operation 1.4.5.2) and the resulting Un-Tested Code is re-tested (operation 1.4.5.3). The result of this phase is a tested version of Optimized Code.

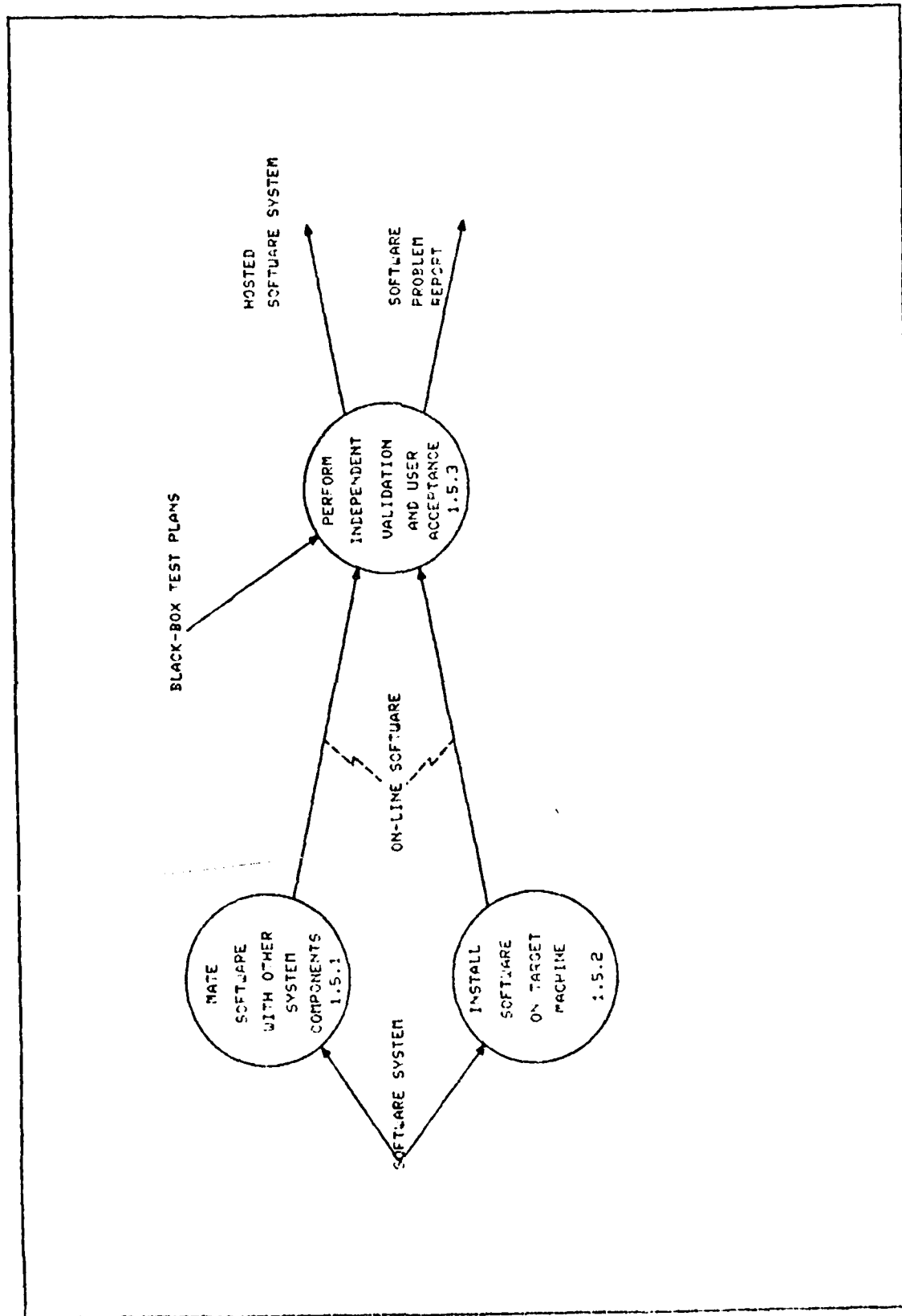


Figure 18: Integrate to and Validate on Target Machine



#### 2.4.14 Integrate to and Validate on the Target Machine.

The integration of the Software System involves either the Mating of the Software with the other Components of it's host system (operation 1.5.1) or the installation of the Software System on to the Target Machine (operation 1.5.2). The result of either of these operations is an On-Line version of the Software. This version of the Software is then tested with the independently developed Black-Box Test Plans (operation 1.5.3) and User Acceptance of the Software is achieved. If errors are detected during this operation, Software Problem Reports are issued, otherwise a Hosted Software System has been achieved.

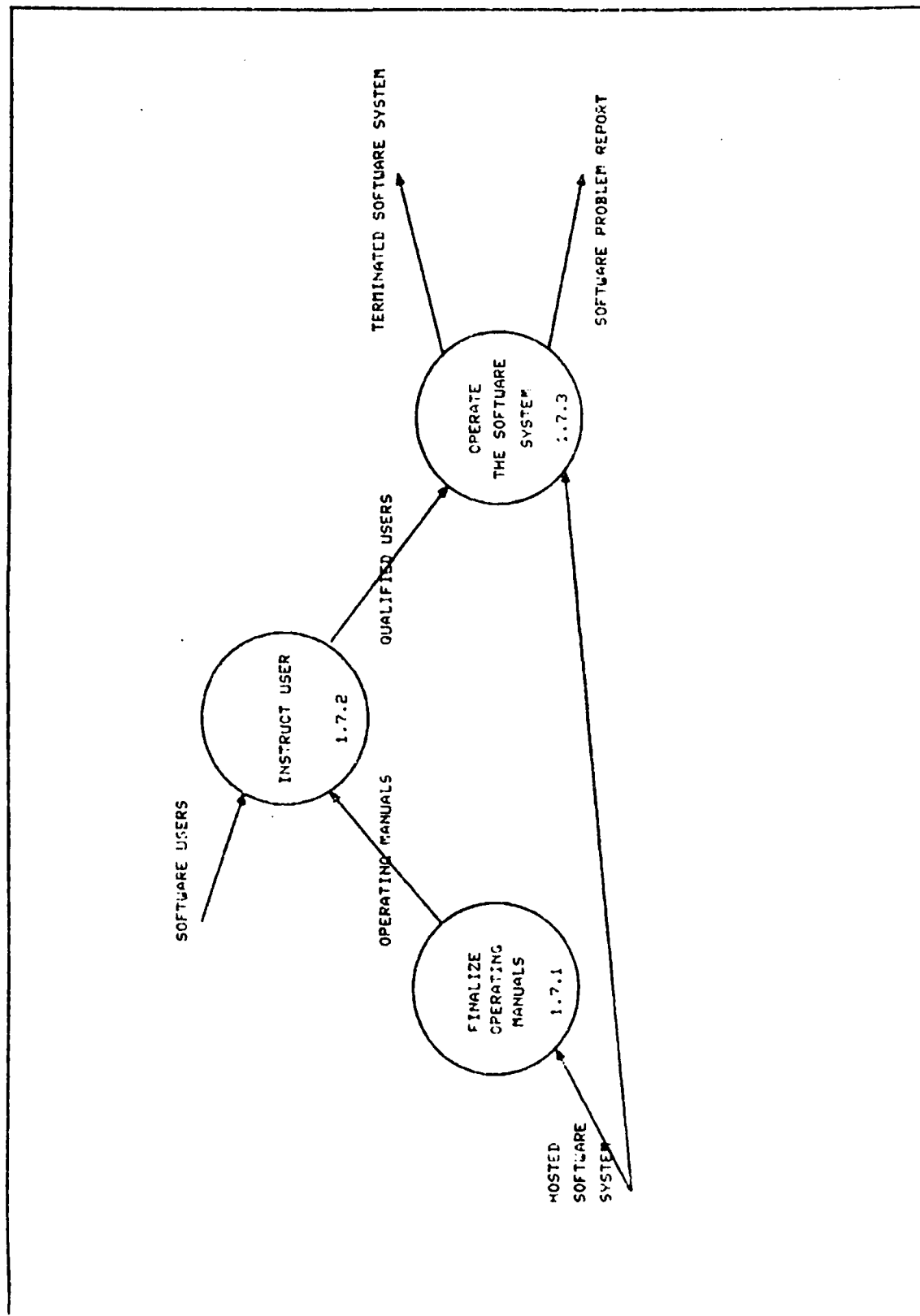


Figure 19: Maintain and Operate Software System

2.4.15 Maintain and Operate Software System. Once the Hosted Software System has been delivered, the Finalization of the Operating Manuals takes place (operation 1.7.1). These Operating Manuals are used to Instruct the Users of the Software System (operation 1.7.2). The result of this operation is a set of Qualified Users. These Qualified Users operate the software (operation 1.7.3). If errors are detected in the software or additional requirements for the software are realized, a Software Problem Report is issued. Once the software system is no longer of operational usefulness, it is archived as a Terminated Software System.

## 2.5 SDW Evaluation Parameters and Criteria

In order to measure the success of any software system in satisfying its requirements, a set of evaluation parameters and a criteria for these parameters must be established prior to implementation and measured following implementation. The evaluation of the SDW is a rather subjective matter. Thus, the criteria for the evaluation parameters is also rather subjective. Evaluation parameters for the SDW are established for two different levels of the SDW implementation. Evaluation parameters for the system level of the SDW determine the SDW's merit as an integrated software development environment. Evaluation parameters for the tool level of the SDW help determine which components of

the SDW should be kept, which should be modified, and which should be discarded.

There are several system level evaluation parameters for the SDW. The first is the average time spent in learning how to effectively use the SDW. This parameter varies with the individual user and the type of development effort he is involved with. The general and subjective criteria for this parameter is minimization, most probably in the range of five to ten days. Another evaluation parameter for the SDW is the level of integration achieved as measured by the life-cycle methodologies supported by the SDW using a sequential application of tools that share data. The initial criteria for this evaluation parameter is the support of the methodologies taught in the AFIT Software Engineering course. The time and effort spent on the software development, as well as, the reliability and quality of the software produced is a third evaluation parameter. The criteria for this parameter is to lessen the time and effort spent on the development while improving the reliability and quality of the software as compared to estimates of these parameters if the SDW was not utilized. A fourth system level evaluation criteria for the SDW is how easily the software product can be updated in response to detected errors or new requirements. The criteria for this parameter is also an estimate of the difficulty involved in the activity if the SDW had not been utilized.

A separate set of evaluation parameters and criteria exist for the tool level of the SDW. The first parameter at this level is the time required to learn the function and operation of the particular tool. The criteria should be 1 to 5 days depending on the complexity and usefulness of the tool. The next parameter is the user's response to the usefulness of the tool in his development effort. As long as some users find the tool useful in their development efforts, the criteria for this parameter is met. The final parameter for the tool level of the SDW is the quality of the tool's output. The criteria for this parameter is determined by the type of the tool. If the tool is used for notational purposes, the output must be of a high enough quality to be included in a formal manuscript. If the tool is used to detect errors in the development, it must demonstrate some level of effectiveness in achieving its goal.

The preceding discussion of the evaluation parameters and criteria for the SDW is not meant to be exhaustive, but rather just a guideline to assist in calculating the benefit of the SDW and in pointing out areas of future improvement.

## 2.6 Summary

The AFIT Software Development Workbench specified in this chapter is to be the realization of an automated software development environment that interactively assists the software developer in producing highly reliable and maintainable software. Capabilities to assist in software development management and to simplify the production and maintenance of many varieties of software production help to reduce the high costs associated with software development. Specific concerns such as integration, user-friendliness, flexibility, testability, etc... are fundamental to the development of a useful software development environment. These concerns are used to direct the design and implementation of the SDW. The previously stated functional model of the SDW demonstrates that the SDW must support software development in all of its development stages and be able to do so in a variety of ways. Finally, a set of evaluation parameters and criteria is established to aid in the analysis of the SDW upon initial completion.

The preceding statement of requirements for the AFIT SDW is purposely limited to higher level requirements. This is done because many of these high level requirements are satisfied by existing software packages. The required functions that are not adequately satisfied by existing packages must be provided by software packages that are

AD-A124 872

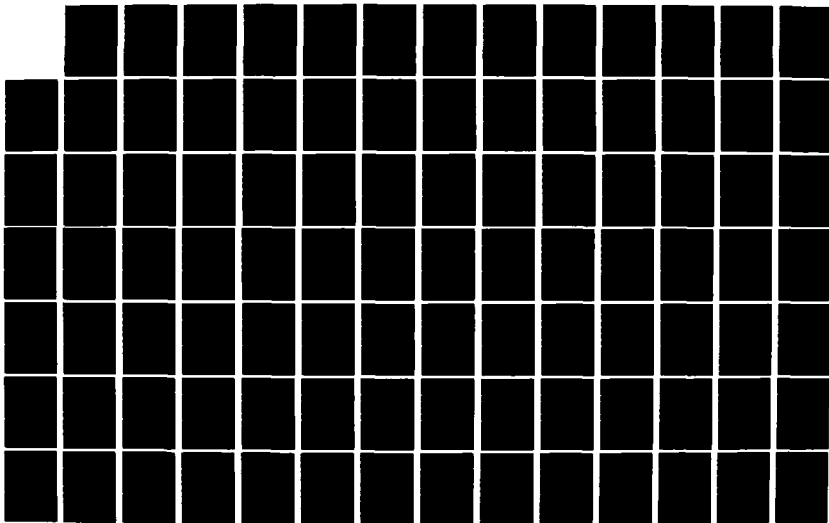
AN INTERACTIVE AND AUTOMATED SOFTWARE DEVELOPMENT  
ENVIRONMENT(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING S M HADFIELD DEC 82  
AFIT/GCS/EE/82D-17

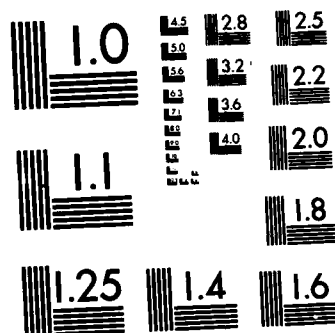
2/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



achieved by a recursive application of the Software Development Life-Cycle.

The concerns and objectives of the SDW, as well as the functional model, stated in this chapter formulate the Requirements Definition Document for AFIT's Software Development Workbench. The goals of this Requirement's Definition Document are very optimistic. The design and realization of these goals is planned to be accomplished incrementally. Initial designs and implementations of the SDW are limited in order to achieve some degree of operational status as soon as possible. The experience gained from using the initial versions of the SDW provides for later improvements to the design and implementation.

### CHAPTER 3: SDW PRELIMINARY DESIGN

### 3.1 Preliminary Design: Introduction

Within the Software Development Workbench (SDW) development effort, the term "preliminary design" refers to the software development stage during which the functional framework of the software system is developed (Ref 90). The preliminary design of a software system is much like the structural framework of a building. The walls and braces that compose the framework of the building are built first with the details of the building's interior left for later. The interior of the building is where most of the activity (for which the building is being constructed) is to take place. However, without a sound framework, the building is not able to support the activities it was designed for. Likewise, the preliminary design designates the framework within which the functional algorithms for the software system are to operate. Without this structure, the functional algorithms would not be able to become a useful part of the entire software system. Thus, the establishment of a sound framework for the software system is the objective of the preliminary design stage.

Just as the constructor of a building has a standard to use in building the framework of their buildings, the software engineer has a standard to use in developing the preliminary design for his software. This standard is most usually a hierarchical framework of managerial and

functional modules. This framework begins with a single high level module at the top which calls/uses other modules. These modules many, in turn, call/use still other modules. Some modules are tasked with managing the lower modules, while the other modules are tasked with other functions required by the software system. The modules are linked together by calls or usage relationships. They may also pass data or control information back and forth. Each of the modules are defined in terms of their purpose or function as well as their inputs and outputs.

The result of the preliminary design stage is a Preliminary Design Document. This document usually includes the a two-dimensional graphic representation of the hierarchical framework of the software system with all relationships and information interfaces between the modules stated explicitly. Structure Charts and HIPO diagrams are common tools used for portraying the software's structure (Ref 90:50,139).

The Preliminary Design Document may also contain other information about the software system. A configuration model is often included in this document. The configuration model specifies the software system and the hardware, data bases, and other components that are required for the software's operation. Besides identifying the individual hardware, software, data base, and other components, the

configuration model also illustrates the interconnections of these components.

The Preliminary Design Document must state how the requirements, previously stated, are to be satisfied by the software system. This is usually done at a high, rather abstract level during the preliminary design stage and refined by the following detailed design stage. Ideally, the satisfaction of the requirements is accomplished by tracing each of the requirements to a module of the preliminary design's hierarchical framework.

The preliminary design for the SDW includes all of the above stated features of a preliminary design. The SDW is a major development effort that is only being initially addressed in this thesis. Anticipated later development on the SDW requires that an evolutionary design strategy be established. This evolutionary design strategy is provided as a guide for these later development efforts. A configuration model of the SDW is developed to establish how the various component hardware and software systems and data bases are to interface with one another. A high level description is provided to state how the objectives and concerns of the requirements definition chapter are to be resolved by the SDW development effort. This includes a "by function" identification of the tools required by the SDW. Finally, a Structure Chart model is developed of the

framework for the SDW. This model emphasizes "functional cohesion" for each module and "data coupling" between the modules (although some control coupling is also required). Functional cohesion and data and control coupling are defined in the SDW Development Data Dictionary included as Attachment 2. The preliminary design of the SDW includes these sections in order to provide a sound framework, not only for the initial SDW development, but also for later follow on efforts.

### 3.2 The Evolutionary Design Strategy

The SDW is designed as a software development environment to serve the AFIT software community. The accomplishment of the SDW objectives requires an extensive development effort that will span many thesis investigations. The experienced gained in each of these follow-on investigations provides for enhancements to the SDW in later follow-on efforts. Thus, the SDW takes on an evolutionary nature. This evolutionary nature is visible through the study of two SDW parameters. One parameter is the number and functional variety of tools. This parameter measures the degree to which the SDW achieves the goals of a "tool kit" approach. The "tool kit" approach, as defined in Chapter 1, is characterized by many single function and distinct tools. The benefit of this approach is that it

provides for a great deal of flexibility in supporting software development. The other parameter used to measure the evolution of the SDW is the level of integration of the resident tools. The level of integration is of concern in the "job shop" approach to software development environments. The "job shop" approach emphasizes a smaller tool set that is highly integrated. These individual tools can be interfaced to provide automated support for life-cycle methodologies. The benefit of the "job shop" approach is that it provides support for and enforcement of software development methodologies. In theory, both of these approaches can be supported by a single software development environment. A large and varied number of tools within the environment would provide the flexibility of the "tool kit" approach, while the integration of a selected subset of these tools would accomplish the "job shop" approach goals. The eventual objective of the SDW development is to achieve the goals of both of these approaches.

The accomplishment of both the "tool kit" and the "job shop" goals in the SDW development is very optimistic, especially when considering the other objectives and requirements of the SDW. For this reason, an evolutionary design strategy is established to allow for the gradual realization of the "tool kit" and "job shop" goals. This Evolutionary Design Strategy is described by the two figures

below.

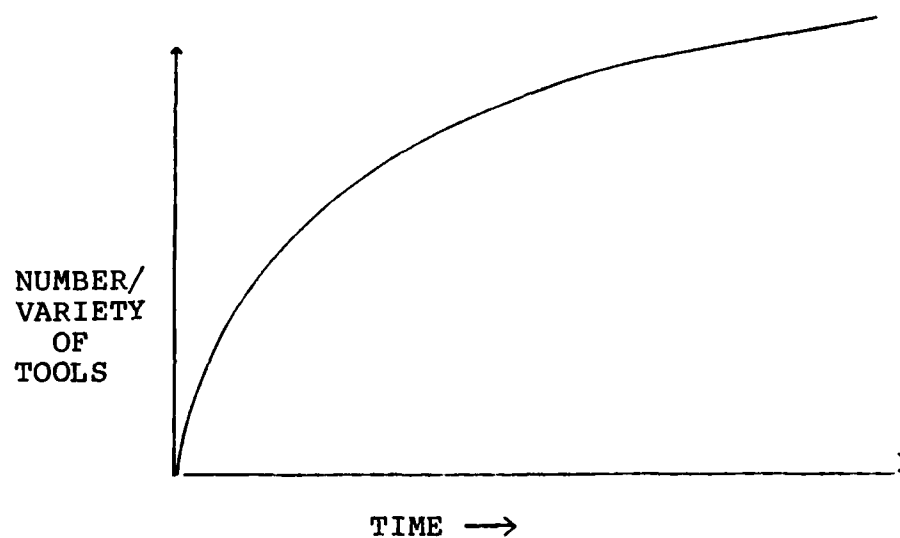


Figure 20: Tool Variety Progression Plan

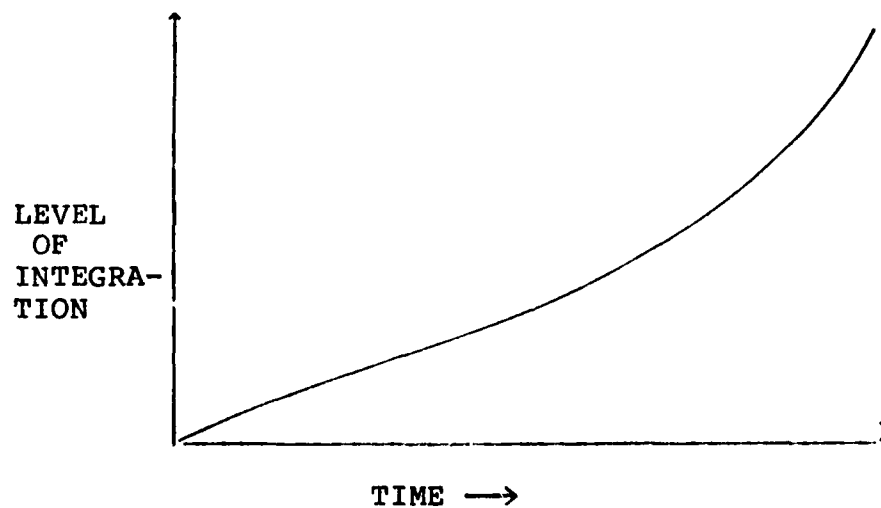


Figure 21: Tool Integration Progression Plan

Figure 20 illustrates that the initial development of the SDW



is to be involved with the inclusion of many component tools in to the SDW. The acquisition of additional tools then levels off as the required SDW functions are satisfied. During the intensive initial acquisition of tools, the level of integration between tools is not emphasized, as illustrated by Figure 21. However, after a comprehensive tool set has been obtained, the level of integration within the tool set is stressed.

The justification behind this evolutionary design strategy is that many of the tools to be incorporated into the SDW are already in existence as stand-alone systems. The early incorporation of these tools into the SDW allows the SDW to become operational very early in the development cycle. This benefits the continued development of the SDW in two ways. First, by using and analyzing these existing tools within an operational version of the SDW, the familiarity with the tools that is achieved aids in the later integration of the tools. Secondly, the experiences of the SDW users with these tools can assist in evaluating which tools should be kept, replaced, or discarded. Thus, the integration of the tool set occurs only after the tool set composition has stabilized. The concept of an evolutionary design is by no means unique to the SDW development effort. Evolutionary design is a characteristic of many software development projects. In fact, William Riddle of the University of Colorado in Boulder recommends

that all software development environments be developed using some type of evolutionary design strategy (Ref 67). His justification for this advice is that the user's requirements for the environment may not be fully understood until some experience with an operational environment has been obtained. Furthermore, if the tools to be included in the environment are being or have been developed outside of the environment's development, their usefulness within the environment must be evaluated.

### 3.3 SDW Configuration Model

3.3.1 SDW Configuration Model Objectives. With the objectives of the SDW Preliminary Design stated and an evolutionary design strategy established, a model of the SDW components and their inter-relationships is developed. This model is referred to as the SDW Configuration Model and it defines the overall structure of the SDW. The objectives of this model are to establish the systemic structure for the SDW in terms of all of its hardware, software, and data base components. The model also identifies all of the data and control interfaces between SDW system components. The SDW Configuration Model is illustrated in Figure 22.

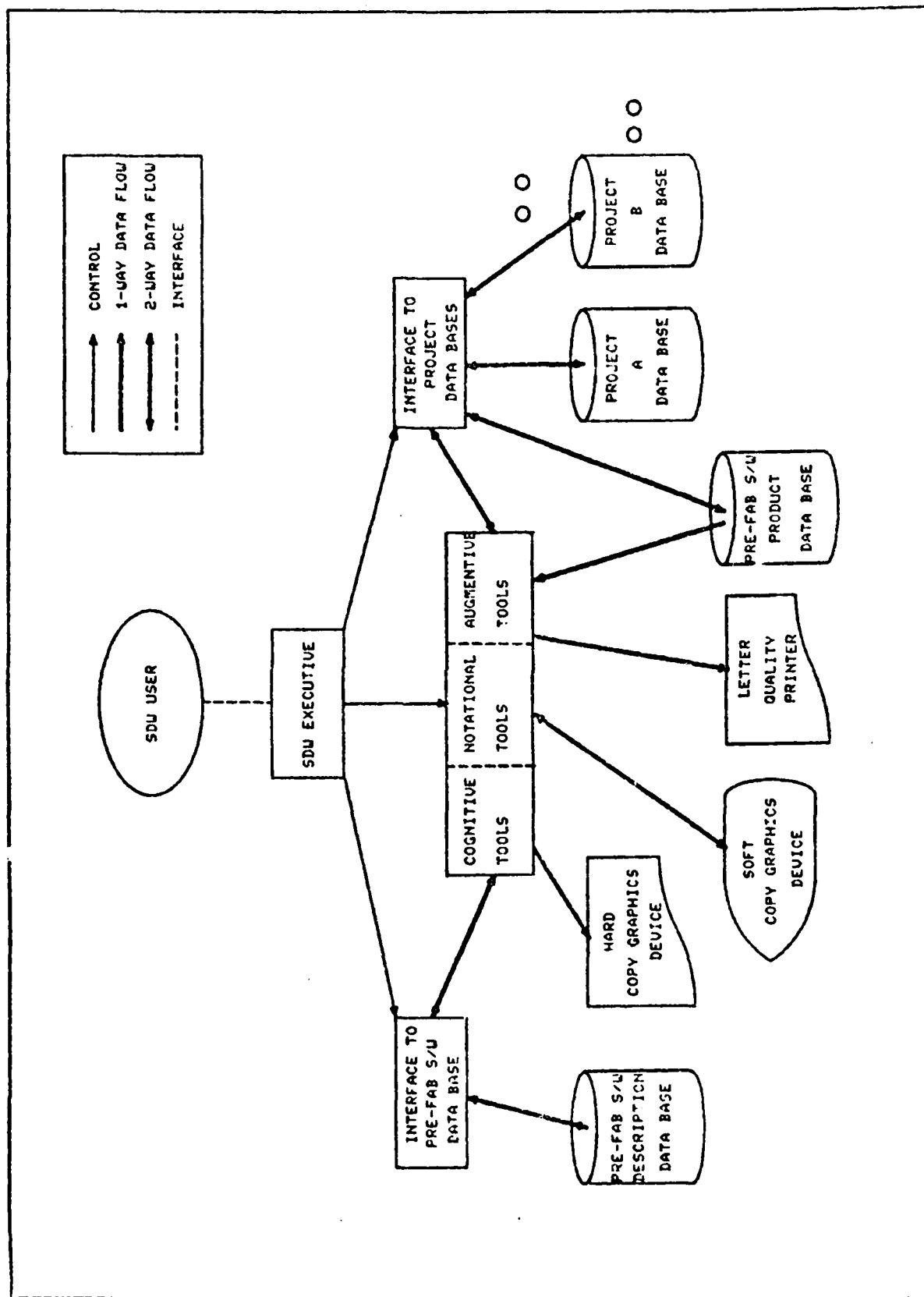


Figure 22: SDW Configuration Model

3.3.2 SDW Configuration Model Description and Justification. The SDW Configuration Model is provided as a framework for the SDW software components and as a guide for the implementation of the SDW. At the top level of the model is the SDW Executive. The SDW Executive is a software component and provides the interface between the SDW user and the SDW itself. The SDW Executive manages and controls all of the other SDW components.

The other software components of the SDW are the Interface to the Pre-Fab Software Description Data Base, the Interface to the Project Data Bases, and the actual automated and interactive tools. The Interface to the Pre-Fab Software Description Data Base allows the SDW user to search the Pre-Fab Software Description Data Base for descriptions and locations of already written software modules and programs that he may require.

The Interface to the Project Data Bases allow both the SDW user and the SDW tools to access the Project Data Bases where all of the software development data and documentation are stored. The Interface to the Project Data Bases also provides for a software transfer link between the Project Data Bases and the Pre-Fab Software Product Data Base, where the already constructed software modules and programs are stored.

The final software component of the SDW is the actual tool set. The individual tools within this tool set are classified by function into three categories. These categories are cognitive tools, notational tools, and augmentive tools (Ref 67). The cognitive tools extend the intellectual capabilities of the SDW user by provided automated and interactive facilities to support Software Engineering methods and techniques. The notational tools assist the SDW user in developing, producing, updating, and maintaining development information. The augmentive tools utilized the computational speed of the computer to check the consistency, precision, and completeness of the development products with great rigor. The SDW tool set is capable of selectively interfacing to other SDW components through data paths. Some tools within the SDW tool set interact with the Interface to the Pre-Fab Software Description Data Base. Most all of the tools communicate with the Interface to the Project Data Bases. Some tools receive software products directly from the Pre-Fab Software Product Data Base. The Notational and Cognitive tools especially use the three input/output (I/O) hardware devices (the hard copy graphics device, the soft copy graphics device, and the letter quality printers).

The remaining components of the SDW Configuration Model include three distinct types of data bases and four hardware I/O devices. The first type of data base is the Pre-Fab Software Description Data Bases. This data base holds the descriptions and locations of existing software packages. This data base assists the SDW user in locating existing software packages that are similar to or may solve part of his particular development project. The Pre-Fab Software Product Data Base is where the actual software packages, described by the Pre-Fab Software Description Data Base, are stored. The last type of data base component in the SDW Configuration Model is the Project Data Base. There exists a separate Project Data Base for each development effort being supported by the SDW. These Project Data Bases hold all of the development documentation and data for the developments being supported by the SDW.

The hardware components of the SDW Configuration Model are all I/O devices. There are four of these hardware I/O devices. They are a hard copy graphics device, a soft copy graphics device, a letter quality printer, and a standard interactive video terminal. The hard copy graphics device is required for the production of on paper copies of the graphical illustrations of the development efforts supported by the SDW. The soft copy graphics device is required for the displaying and editing of these graphical illustrations on a video display. The letter quality printer is used for

producing copies of software development documentation. The high level of quality is required because this documentation must be included in formal manuscripts such as theses. The last hardware I/O device that is identified by the SDW Configuration Model is the standard interactive video terminal, realized in the model as the component labeled User. This component is referred to as User because it is the usual device used to interface with the SDW. Of course, separate terminals are required for each concurrent User of the SDW.

The SDW Configuration Model provides a framework for the developing of the software and data base components of the SDW. These are the components that are the emphasis of this initial development effort of the SDW. However, the model also specifies the required SDW system structure that must be present to satisfy the objectives and concerns of the SDW development effort as stated in Chapter 2. The next section of this chapter (Chapter 3) explains how each of these objectives and concerns are addressed by the SDW design. References are made in that section to the SDW Configuration Model and how its components are used to satisfy these objectives and concerns.

### 3.4 Resolution of the SDW Development Objectives and Concerns

The SDW Configuration Model establishes the baseline configuration of the SDW as viewed as a total hardware, software, and data base system. The configuration model, however, does not state the detail of how each of the specific requirements for the SDW are addressed. In order to specify the mechanics of how each requirement is resolved, each requirement is taken individually and the components of the SDW that address that requirement are established. The first requirements to be resolved are the developmental concerns and objectives of the SDW. They are discussed in the order presented in chapter 2. Each statement of an objective or concern (in chapter 2) is referenced by the paragraph(s) that describes its resolution (in chapter 3). Discussions of how each objective/concern is resolved detail both the initial mechanisms used and those to be incorporated into the SDW in follow-on efforts.

3.4.1 The Reduction of Software Errors (Resolves 2.3.1). Several mechanisms are introduced into the SDW in order to reduce the occurrence of software errors. First, the SDW utilizes a variety of software engineering methodologies supported by interactive and automated tools to encourage software development in accordance with the established software engineering practices and principles. The SDW Preliminary Design calls for such tools to support



activities in all of the software development stages. For requirements definition, the SDW Preliminary Design calls for tools to support the development of Data Flow Diagrams (Ref 90), Structured Analysis and Design Technique (SADT) (Ref 79), and other english-like requirements languages such as the Requirements Statement Language (RSL) and the System Specification Language (SSL) (Ref 2;3). Preliminary design and detailed design are supported by tools for such methodologies as HIPO (Hierarchical Input Process Output) (Ref 90:139), Structure Charts (Ref 90:50), and Structured English. Other methodologies, such as Nassi-Schneiderman Charts and N-Squared Charts may be supported in later versions of the SDW. The implementation and integration stages of development are supported by facilities for developing and recording of top-down implementation and test plans (Ref 90:210). The implementation and integration stages involve a great deal of specialized code testing tools. These tools are discussed in greater detail in the section dealing with "Testability".

While the tools used for testing during the implementation and integration stages do support software engineering testing principles, they are also considered a variety of "augmentive tools". Augmentive tools are the automated tools that utilizing the speed and computational power of the computer to test for completeness and consistency of intermediate and final software products.

These augmentive tools form an important aspect of the SDW in the effort to reduce software errors. Augmentive tools are designed into the SDW to test the products of all stages of software development. These tools enable the developer to test his products with great rigor which allows many errors to be detected very early in the development cycle.

Ultimate plans for the SDW call for all development data to be stored in an unified data base. This would eliminate much of the potential for consistency errors occuring when not all of the development data from different data bases is updated to reflect a requirments, design, or software change. The use of an unified data base is not implemented in the initial version of the SDW. A stabilized tool set for the SDW is required prior to the establishment of the unified data base.

3.4.2 Responsiveness to Change (Resolves 2.3.2). As pointed out earlier, software is a dynamic entity. Errors found during development or operation must be corrected with changes to the software. Changes in the user's requirements also require modifications to the software. Thus, the software development data must be changeable. The SDW supports the modification and updating of development data by utilizing three types of mechanisms. First, all of the SDW components provide for both the building and modifying of their outputs by updates to their inputs. These outputs

are all stored in a common data area. This common data area is the Project Data Base (3.3). Initially, the Project Data Base is simply a directory structure of individual files. Later, the Project Data Base is to be an unified data base that joins these individual files.

The Project Data Base also stores the test cases used to validate the software system. Once a modification has been made, these test cases may be called up to validate the modified software. The final mechanism to facilitate ease of software modifications is the enforcement of a tracing relationship between corresponding components of the different development stages products. An example of this type of mechanism is the tracing of requirements to design and code modules. Such a mechanism would allow a change of requirements, for example to be taken through only the design and code modules that it is related to. Initially this mechanism is done manually with the use of comments in the intermediate products. However, with the use of an unified Project Data Base, the traceability mechanisms could be built directly into the data base conceptual schema. The achievement of traceability is further discussed in 3.4.7.

3.4.3 Rapid Assessment of Design Alternatives (Resolves 2.3.3). The software developer is often faced with design decisions that must be made with little knowlege of the alternatives and their effects on the software system. The

SDW assists the developer in this problem by providing prototyping and simulation tools. These types of tools allow the developer (SDW user) to model his software system very quickly. The model of the system is then run through a simulation of the system load and feedback is produced for the model. The model is then easily modified to reflect another design alternative and the model is again run through the simulation and the produced results are compared. The SDW supports prototyping and simulation in three fashions. First, the SDW provides tools for independent prototyping and simulation. Second, it provides translation interfaces for converting requirements and designs into simulation models. Thirdly, many of the Requirements Definition and Design tools have built-in simulation capabilities. The ideal of the prototyping and simulation capability of the SDW is to provide near real-time feedback on different design alternatives.

#### 3.4.4 Automated Documentation Support (Resolves 2.3.4).

The genesis of the SDW development effort was to provide automated tools to assist in the production of the documentation associated with software development. Automated documentation support is still of major significance. The SDW Preliminary Design calls for the inclusion of many tools to support the various software engineering methodologies that utilize two-dimensional graphics. These tools must be capable of producing hard

copy outputs in addition to the video outputs. An interactive graphics editor is also included in the SDW Preliminary Design. This editor allows the SDW user to develop his own customized documentation and even develop his own graphical development methodologies. A text editor, most favorably a screen-oriented text editor, is included in the SDW Preliminary Design for use by the SDW user in creating and modifying the textual documents associated with the development. A word-processor program is also called for to assist in the development of the textual information.

All of these "notational tools" are supported by the four hardware components of the SDW Configuration Model. These components are the video terminal, the letter-quality printer, the hard copy graphics device, and the soft-copy graphics device.

3.4.5 Software Managerial Capabilities (Resolves 2.3.5). Although not addressed by the initial version of the SDW, later investigations could use the SDW as an excellent test bed for developing software managerial capabilities. Such capabilities may be the automating of status reports on an individual software development effort, or the interactive development and maintenance of plans and schedules for the development. The incorporation of software managerial capabilities into a development environment is of extreme importance when the number of

developer/programmers gets much over a half-dozen or so. Such capabilities are not easily developed, but if properly developed they could be of great significance.

Besides the specific objectives of the SDW development, the requirements definition chapter (Chapter 2) establishes a set of concerns that must be addressed by the development. The following paragraphs provide explanations of how these concerns are addressed in both the initial and the follow on SDW development efforts.

3.4.6 Integration (Resolves 2.3.6). The first concern listed is that of integration. The term, integration, can be used with a variety of meanings within the discussion of software development environments. Within the SDW development effort, this conflict of semantics is resolved by discussing levels of integration, with each of these levels taking on a distinct meaning.

The highest level of integration is simply that all components of the SDW are located on a single machine. This level is achieved by hosting all of the SDW components on the target machine (the DEC VAX 11/780). The next level of integration deals with how the SDW components are accessed. A single common interface is provided to the SDW components through the SDW Executive illustrated in the SDW Configuration Model (3.3). The third level of integration is the use of a common data storage area for all of the

development data from a single development project. This is achieved using the Project Data Bases, again illustrated in the SDW Configuration Model. Within the Project Data Bases, a separate and independent schema exists for the data from each of the development stages.

The last two levels of integration are by far the most interesting, however, they are left to be implemented by later SDW follow-on development efforts. The first of these last two levels is the actual integration of specific components of the tool set. This could be done either by interface routines that reformat the output of one tool to be the input of a next tool or by the design and implementation of an original tool set that is integrated by design. The first approach, using the interface routines, is probably the easiest to realize given the evolutionary design strategy (3.2). However, integration by design has proven most effective as it has been realized in the University of California at San Diego's P-System (Ref 87) for the development of software written in Pascal. The P-System uses a similar command syntax for all of its components and allows its different tools to call each other. The ultimate goal of this level of integration is to enforce consistency of intermediate and final development products through strict tool set integration. This is accomplished by using previous development products as constraints on the production of later products. The

mechanisms used for accomplishing this are discussed in the section on Consistency and Completeness (3.4.13).

The second of these last two levels of integration involves the use of a shared data base to hold all of the development data for a single project. This type of data base would realize the relationships between each of the separate data schemas that exist for each of the development stages. The use of this shared data base is further discussed in the next section that deals with traceability, which is the major benefit of this level of integration.

3.4.7 Traceability (Resolves 2.3.7). Traceability refers to the use of relationships or mappings to track between units of the products of each of the different development products. In particular, traceability involves the mapping of requirements specifications to design units, of design units to code modules, and code modules to updates of that code module. Changes to the software are easily maintained using these mappings.

In the initial version of the SDW, traceability is handled manually by referencing requirements in the design specifications and the design specification in the comment areas of the code. Later versions of the SDW are to use integration mechanism to achieve an automated traceability capability. The particular integration mechanism is a shared data base. This shared data base is actually a



distinct schema added to the top of the Project Data Bases referenced in the SDW Configuration Model. This added schema is referred to as a Common Data Model (CDM). The schema defined as the CDM establishes and preserves all of the relationships between the different schemas of the Project Data Base. By utilizing these stored relationships, the mappings between requirements, design, implementation, and maintenance units are automatically provided for traceability.

#### 3.4.8 User-Friendliness (Resolves 2.3.8).

User-Friendliness is a fundamental concern of the SDW development because of its direct influence on the eventual acceptance of the SDW by the AFIT software community. Menus are provided at each level of the SDW Executive, as well as help files that are integrated into the VMS Help facility. A primary criteria for the selection of specific tools during the detailed design stage is the tool's user interface. Automated, on-line teach capabilities are designed into the SDW to assist the new users. Fail soft error recovery (Ref 60) is provided for in the design so that users are not left helpless due to an erroneous input. As the state-of-the-art in artificial intelligence, speech synthesis, and pattern recognition advances, later version of the SDW may utilize an intelligent talking front end. This would allow SDW users to develop software by conversing with the SDW in normal english.

3.4.9 Testability (Resolves 2.3.9). Utilizing the potential of the computer to help detect development errors is a major emphasis of the SDW for the achieving of the SDW objective to help eliminate software errors (3.4.1). The augmentive tools referenced by the SDW Configuration Model are included in the SDW Preliminary Design for this purpose. They provide automated and interactive mechanisms to test and validate the products of each step of the software life-cycle. During the requirements definition and design stages of software development, the augmentive tools are built into the cognitive tools. Some simulation capabilities are included to round out the validation functions for these stages. A variety of specialized testing and validation tools exist to support the implementation, integration, and maintenance/operation stages of the software life-cycle. The specific types of these tools that are incorporated into the SDW Preliminary Design are listed and explained below.

- Static code evaluators perform code evaluation that does not require the execution of the code (Ref 12:42). The purposes of this type of testing are the insuring of:

- Consistent language usage
- Consistency of redundant information
  - Type declarations
  - Physical dimensions
  - Assertions

- Consistent variable setting and usage
- Consistent code structuring
- Dynamic code structure evaluators require the actual execution of the code (Ref 59:7). Three general types of these evaluators are commonly recognized.
  - Execution monitors check for and stop at error conditions (in a manner similar to the Ada Exception Handling capability (Ref 1)).
  - Software monitors allow for the stating of assertions in the code and the testing of those assertions during execution.
  - Dynamic debuggers provide trace and dynamic code and data updating capabilities.
- Code instrumenters provide means for collecting data, either conditionally or unconditionally during the execution of the software (Ref 12:40).
- Test case generators provide for automatic or semi-automatic production of input data (Ref 12:43).
- Symbolic execution tools analyze software along a given path within the software and determine a set of input data that causes that path to execute (Ref 61)
- Test analyzers are algorithms for estimating the degree of "testedness" of a program (Ref 12:44).
- Performance monitors insert additional code into the software that calculates the time spent in each software module to locate the areas where the greatest gains from optimization can be realized (Ref 89).

#### 3.4.10 Pre-Fabricated Programming (Resolves 2.3.10).

pre-fabricated programming is a term coined to refer to the use of existing software systems and modules to satisfy requirements and design specifications within developing software systems. The use of pre-fabricated programming can

significantly benefit many software development efforts by eliminating the need to design, code, and test many functions and subfunctions. The SDW Preliminary Design uses the Pre-fab Software Description Data Base and the Pre-Fab Software Product Data Base to support the concept of pre-fabricated programming. These two data bases are referenced by the SDW Configuration Model in section 3.3.

The Pre-Fab Software Description Data Base records the existence of many existing software modules and programs. A description of the software unit, an associated list of keywords, the author(s), and the software unit's location are stored in this data base for each existing piece of software. The Pre-Fab Software Product Data Base is simply a collection of the actual software units (in either code, design, or both). The Pre-Fab Software Product Data Base may be distributed over many locations with some units being resident on system disks and some archived on other disks, tapes, cards, etc. More detailed descriptions and models of these data bases are provided in the next chapter on detailed design.

3.4.11 Support the Entire Software Life-Cycle (Resolves 2.3.11). A major principle of software engineering is the view of software as possessing a life-cycle. The SDW is required to support this entire life-cycle. This concern is addressed in two fashions. First, the SDW tool set is

design to provide capabilities to support each of the software life-cycle stages. Second, the SDW stores all of the development data associated with each of the life-cycle stages in a common data area for each development effort, the Project Data Bases.

3.4.12 Flexibility (Resolves 2.3.12). The SDW is an evolving environment that is designed as both a pedagogic tool and a creative tool, since the development of software is a creative effort. Thus, a certain degree of flexibility is required to support the SDW as a creative tool. In order to achieve this flexibility, the SDW provides for an evolving tool set. The SDW Executive is design to be easily modifiable to incorporate new tools or discard old ones. By utilizing the "tool kit" approach to the initial SDW development (section 3.2), there is no set order or restrictions on the operations of the different SDW tools. The initial design of the Project Data Bases is a loose file structure that is easily updated to support new tools. Language-specific tools are provided for a variety of programming languages to provide the SDW use flexibility in choosing a language. Many of the SDW component tools utilize languages and conventions that are expandable to fit individual requirements. Finally, an interactive graphics editor capability is designed into the SDW to allow SDW users to create their own original graphics and graphical methodologies.

#### 3.4.13 Consistency and Completeness (Resolves 2.3.13).

During the development of large and complex software, the software developer may quite easily forget or overlook the specification of some development details. Additionally, several references to a particular software detail may not of been made consistently amidst the complexity of the entire system. Such problems are not easily found by the human eye, however, a computer could exhaustively search for such problems in a much more economical fashion. The augmentive tools previously mentioned in the SDW Configuration Model are used by the SDW to insure consistency and completeness of the software development products. Most of the augmentive tools used to check for consistency and completeness are built into the cognitive tool that produces the particular development product that is being analyzed. These augmentive tools thrive off the same sub-schema of the Project Data Base that is used by the associated cognitive tool. Some of the development details that are analyzed by these augmentive tools are the consistency and completenss of functional requirements specifications, of design specifications, and of code modules. The explicit interfaces between code and design modules are checked by interface checkers. These tools check the number, type, and order of parameters that are passed accross the interfaces. A units checking capability is also employed to insure that assignment statements are

dimensionally correct.

The previously mentioned mechanisms deal primarily with internal consistency and completeness checking. External consistency and completeness checking insures the proper translation of development data between the different development stages. Interfaces between the SDW component tools use the output of one tool to constrain the input of the next tool. Consider for example the development of a design for a software system. The requirements are already specified. A syntax-directed editor is used to state the design in some particular design language. The syntax-directed editor accepts both the Backus Normal Form (BNF) description of the design language and the previously stated requirements as inputs. Using these inputs, the editor only permits the creation of design specifications that are consistent with both the design language rules (the BNF description) and the stated requirements. An attempt to state a design specification that is not traceable to a requirements specification is either not allowed until the requirements document has been updated or is allowed with the system automatically updating the requirements document to reflect the design addition. If the second alternative is used, the update to the requirements document is recorded as an unapproved requirements modification. This helps the developers and users to identify where modifications to the requirements have been made. A similar type of scenario can

be employed when moving from the design specification into the actual code. Such a capability is a major advance in maintaining consistent development documentation.

3.4.14 Explicitness and Understandability. The need for explicitness and understandability is common to many aspects of the SDW. The SDW components utilize graphics extensively to improve the understandability of the produced documentation. Furthermore, the languages used to provide inputs for the tools are selected based upon a criteria of both explicitness and understandability.

3.4.15 Documentation Support (Resolves 2.3.15). The resolution of the requirement for automated documentation support is addressed in a previous paragraph (3.4.4). The requirement for automated documentation support is addressed both as an objective and a concern of the SDW development. As a concern of the SDW development, automated documentation support is resolved by the inclusion of software notational tools and hardware output and storage devices into the SDW Preliminary Design. The production of high quality documentation and the archiving of that documentation for later reference are also specific requirements of the AFIT software community (refer to 2.3.19).



3.4.16 Updateability (Resolves 2.3.16). The commonly realized dynamic nature of software dictates the requirement for updateability within the SDW. Updateability is achieved by designing the Project Data Base to store both previous and current versions of the development data. The notational and cognitive tools are utilized to actually modify the data. The notational tools include the text and graphics editors.

3.4.17 Language Independence (Resolves 2.3.17). As stated in the SDW requirements (2.3.17), the selection of a particular programming language for a development effort is not required until the Detailed Design or the Implementation stage of development. For this reason, all of the SDW component tools that support pre-implementation activities are programming language independent. These tools may utilize specific specification or design language to operate, but the tool specific languages do not limit the choice of application language for the eventual implementation.

3.4.18 Early Prototyping (Resolves 2.3.18). By prototyping a software system very early in its development, design alternatives can be analyzed. Furthermore, user experience with the prototype can help to drive the design. The SDW uses stand-alone simulation tools to achieve the requirement for early prototyping. However, the simulation

tools may be driven by the requirements and designs develop by the cognitive tools. This is facilitated by translation routines that convert requirements statements and designs into simulation models that prototype the software system.

3.4.19 AFIT Specific Objectives and Concerns (Resolves 2.3.19). Besides being a general investigation of software development environments, the SDW development effort is directed at achieving an operation environment for the Air Force Institute of Technology (AFIT) software community. As a result, the SDW development effort specified the particular requirements of the AFIT software community for the actual SDW. The AFIT users of the SDW are categorized in two groups. One set of SDW users are those students enrolled in the software engineering course (EE 5.93). The other set is the faculty and thesis students using the SDW for major software developments.

Students in the software engineering courses use the SDW as a pedagogical tool for learning the classical principles of software engineering that are supported by the cognitive tools of the SDW. Facilities for on-line teaching are designed into both the cognitive tools and the SDW Executive.

The faculty and thesis students that compose the rest of the SDW users have a different set of specific requirements that are resolved by the SDW design. The many separate projects of this set of users are handled in a secure manner by using the distinct Project Data Bases illustrated in the SDW Configuration Model (3.3.2). The Project Data Bases also provide for the archiving of development data if the development is a continuing one. The last specific requirement for this class of users is the need for high quality documentation support. This is achieved by the hard copy graphics device and the letter-quality printer specified in the SDW Configuration Model.

### 3.5 SDW Structural Model

The objective of the SDW Structural Model is to illustrate the hierarchical compositions of the SDW components into a functional environment. The individual components of the SDW are included into the structural model in order to satisfy the requirements for such components as stated in the SDW Functional Model (2.4).

#### 3.5.1 Methodology Utilized for the Structural Model.

Three separate design techniques are candidates for illustrating the structural model of the SDW. These

techniques are IBM's HIPO (Hierarchy plus Input Process Output), Higher Order Software's HOS technique, and the classical Structure Chart technique. Each of these techniques utilize a hierarchy of design modules. They each specify the inputs and outputs of each module (sometimes referred to as a function) as well as a titles of modules. The differences in the techniques are realized at the more detailed levels of illustration.

The HIPO technique uses a special digraph called a tree to illustrate it's Function Chart (Ref 90:139). An example of a Function Chart is realized in Figure 23.

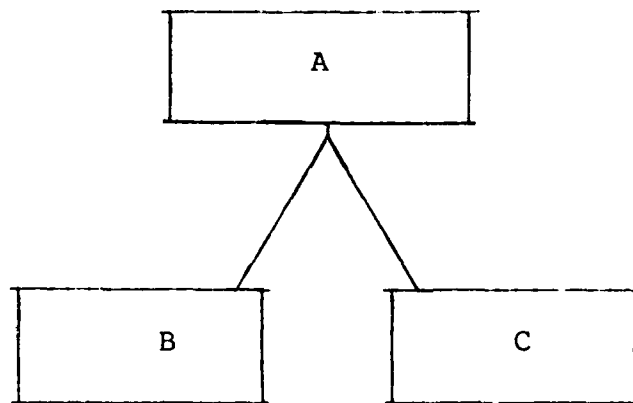


Figure 23: Sample HIPO Function Chart

The root of the tree is the main module that calls all of the other modules either directly or indirectly. In Figure 23. module A is the main module and it calls modules B and C. Each module is described in more detail by the IPO chart

that specifies the exact inputs, processes, and outputs of each module. The IPO (Input Process Output) diagrams use three block as shown in Figure 24.

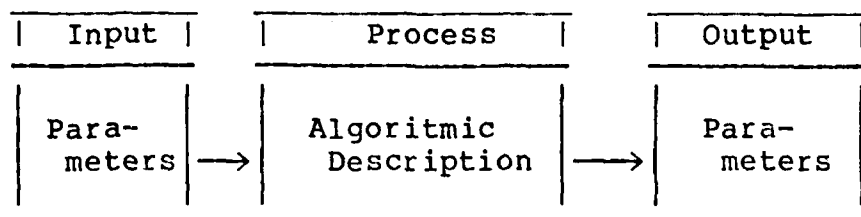


Figure 24: IPO Diagram Sample

The HIPO technique does not, however, specify an ordering or conditions on the calling of subordinate modules nor does it specify the passing of parameters between modules. The HOS technique originated to aid in the development of software designs for the NASA's Apollo and SkyLab programs (Ref 34:72). The HOS technique utilized a hierarchical structure of mathematical functions. This structure is similar to the Function Chart of Figure 23., however each box represents a function with the inputs to the function state immediately to the left of the box and the outputs immediately to right. This modified use of the boxes is illustrated in Figure 25.

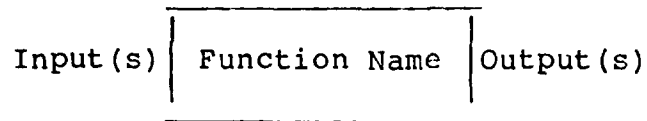


Figure 25: HOS Function Specification

The HOS technique uses an implied left to right ordering of subordinate modules. This ordering may be augmented by special codes at the immediately superior function that specify if the calls are conditional, iterative, or recursive.

The HOS technique does not, however, differentiate between control and data parameters. The Structure Chart method (Ref 90:141) that has been utilized since the late 1960s does differentiate between control and data parameters. The Structure Chart method also utilizes the basic Function Chart hierarchy shown in Figure 23 and has conventions for illustrating conditional and iterative calls of modules. Parameters are shown as vectors with inputs to a module pointing down to the subordinate module and outputs pointing back up to the calling module. Data parameters are shown with vectors that originate with an unshaded circle. The vectors illustrating control parameters originate with a shaded circle. There is no implied ordering to the subordinate modules, however, the proper input parameters to

a module must exist before the module can be called.

The HOS and Structured Chart techniques are especially useful in illustrating the parametric relationships between modules. However, this facility is not of great significance to the SDW Structural Model. The HOS technique views components in terms of functions. The components of the SDW are tools and aids which are not properly expressed in terms of mathematical functions. Of the three candidate techniques for the SDW Structural Model, the HIPO technique seems best suited because of its treatment of modules as distinct units invoked by other units. The components of the SDW are best described as such distinct units because many of them exist as stand-alone systems already.

The HIPO technique does have its drawbacks in terms of the SDW Structural Model. The major problem is that the IPO part of the technique describes the process associated with the module in a psuedo-algorithmic manner. The components of the SDW need only to be described in general terms at the level of detail involve in this preliminary design. The IPO part of the technique is thus replace by a simple formatted textual module specification. This module specification includes the title or type of the component tool, the calling module (tool), any subordinate modules (tools), the inputs and outputs, a functional description of the tool, a comment area, and a special entry that traces the module

(tool) to the specific requirement(s) that it satisfies.

3.5.2 The Actual SDW Structural Model. The SDW Structural Model is a specification of the major components of the SDW. Most of these components are designed to aid in the development of software by supporting and enforcing the use of Software Engineering methodologies. However, some of the components are utilities to support the other tools or facilities to aid in the actual use of the SDW. The SDW Structural Model identifies each of these components and illustrates its position in the SDW hierarchy.

The SDW, being a software development environment, is more of a collection of individual tools (that may or may not be integrated) than it is a software system per se. As a result, the calling relationships illustrated by the vectors in the model may not be formal calls as one would realize in most software programs. Rather, they represent that the modules are used by the SDW user as part of the environment and are referenced by the SDW Executive. However, the calling relationships between the subordinate modules do take on the usual meaning.

The actual SDW Structural Model is illustrated in Figure 26. Each of the SDW components referenced in the model are describe in greater detail in Appendix C.



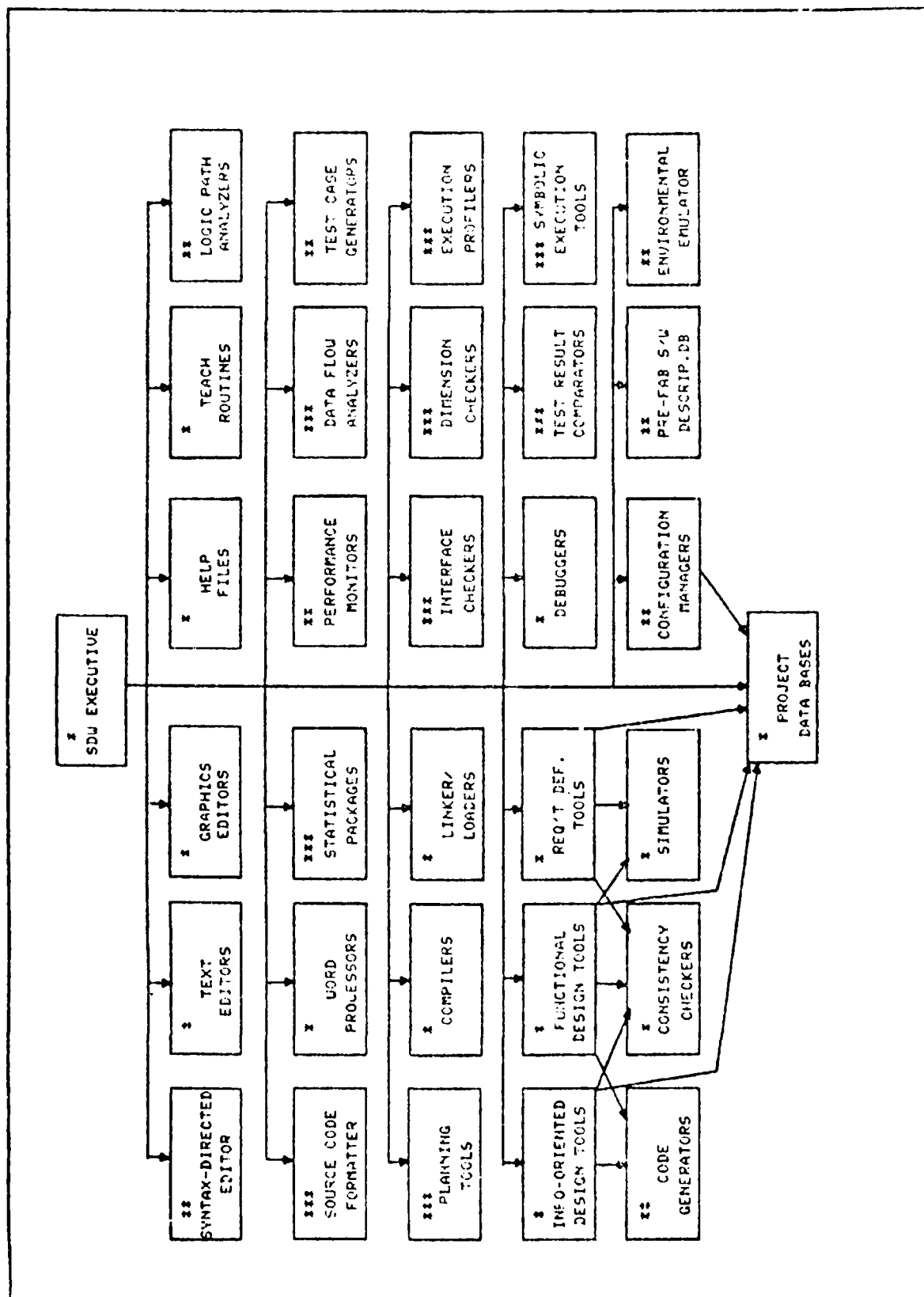


Figure 26: SDW Structural Model

In accordance with the evolutionary design strategy (2.2), the SDW Structural Model is to be implemented incrementally. Each block of the model is marked with one or more asterisks to indicate it's relative significance to the SDW. The semantics of these asterisks are as follows:

- \* -means the module is of immediate importance to the SDW as should be implemented as soon as possible.
- \*\* -means the module is important but not critical to the initial implementation of the SDW.
- \*\*\* -means the module would be nice to have in the SDW but can be done without.

The modules specified with a single asterisk (\*) are the SDW components that are of fundamental importance to the initial implementation of the SDW because they provide the basic development facilities common to most environments, a user-friendly interface and framework for the SDW per se, or aid in the pre-implementation stages of software development. The pre-implementation stages are given greater significance because errors made during these stages tend to be much more expensive to correct (Ref 2).

Modules marked with two asterisks (\*\*) are deemed to be of the next level of importance to the SDW because they do not meet the criteria for primary importance, but are necessary for a truly state-of-the-art environment.

The final set of modules, identified with three asterisks, are those facilities that are nice to have in a development environment but are not of critical significance. The inclusion of these tools into the SDW does not effect the further investigation of the advanced topic associated with the SDW project.

### 3.6 Summary

The purpose of this chapter is to define the preliminary design for the AFIT Software Development Workbench. In order to define this preliminary design, an evolutionary design strategy is established. This design strategy calls for the initial versions of the SDW to emphasize the inclusion of many independent tools into the SDW. Later versions of the SDW are to investigate the integration of these tools into specific methodologies according to the design strategy.

A configuration model of the hardware, software, and data base components of the SDW is included in the chapter. This model defines the SDW as a multi-faceted system.

The many objectives and concerns of the SDW development are resolved with specific approaches to the development effort both initially and for later efforts.

The culmination of the preliminary design is the presentation of the SDW Structural Model. This model illustrates the many components of the SDW, describes the components, and specifies the significance of the component to the SDW.

The preliminary design that is presented in this chapter is a guideline for the continuing development of the SDW. As a guideline, it is carefully orchestrated to resolve the requirements, objectives, and concerns of the SDW development. The rest of this thesis investigation focuses in on particular aspects of the SDW. The justification behind this narrowing of scope is to allow the initial implementation of the SDW within the time frame of this investigation. The next chapter deals with the detailed design of several components of the SDW. That chapter identifies specific existing and available tools to satisfy some of the design specifications of this chapter.

CHAPTER 4: SDW Detailed Design

#### 4.1 Introduction

The detailed design stage of the software life-cycle deals with the development of the functional algorithms required by each design module specified during the preliminary design (Ref 10:7). In larger developments, some of the design modules of the preliminary design may be extensive sub-systems. The detailed design of this type of module involves a recursive application of the software life-cycle. That is, the requirements for that design module must be explicitly defined, a sub-system Preliminary Design developed, and then the algorithms for that preliminary design may be stated.

When implementing a top down implementation and test plan, the Detailed Design stage often overlaps with the next stage, implementation. The top level modules are often designed in detail and coded prior to the algorithmic design of the lower modules. This strategy is followed because testing of the implementation of the top level modules may reveal the need for modifications that affect the lower modules.

The algorithms developed during the detailed design stage must be both concise and precise. In most cases, the algorithms can be developed independent of any implementation language.

The detailed design stage of the Software Development Workbench (SDW) has several important aspects which are described in this chapter. The first is the selection of component tools for Version 1.0 of the SDW. This aspect involves establishing a set of criteria for selection and then making and justifying the selections. The next aspect is the development of the SDW Executive sub-system structure. This development involves the defining of a set of detailed requirements, as well as, a preliminary design and algorithmic design of the SDW Executive. The final aspect is the detailed design for the Project Data Bases files. A simple directory structure is used for the initial Project Data Base design. However, a high level design for later versions of the Project Data Bases is also suggested.

The detailed design stage of the SDW development is a most significant part of this thesis investigation because it deals with the realization of the concepts developed during the earlier theoretical activity.

#### 4.2 SDW Component Selection

The Software Development Workbench (SDW) can be realized as two distinct classifications of components. The first is the Software Development Workbench Executive (SDWE) and the second is the SDW component tools. The selection of

SDW component tools is very fundamental to the effectiveness of the environment to support the development of software. Prior to the selection of component tools, an apriori set of criteria must be established as a basis for the selections. The first part of this section deals with the definition of this set of criteria. The second part specifies and justifies the selections of component tools for Version 1.0 of the SDW.

4.2.1 Selection Criteria for the SDW Components. The selection criteria for the SDW components is realized within three categories. The first of these is the ability of the component option to meet the design specifications established in the SDW Preliminary Design. The second is the availability of the component option to the Air Force Institute of Technology/Electrical Engineering Department. The final is a result of the limited time and resources available to this thesis investigation. Since, only a few new components may be installed on the SDW during this investigation, due to limited installation time and limited disk storage space, some type of ordering criteria must be established for deciding which types of tools must be given first priority for incorporation in to the SDW.

The ability of the component option to meet the design specifications of the SDW Preliminary Design is an obvious criteria for component selection. However, just because it



is obvious, does not mean it could not of been overlooked. For this reason, each of the SDW components selected for incorporation into the SDW must be referenced back to a specific design module specification in the Specification of SDW Preliminary Design Module (Appendix C).

As an academic institution, the Air Force Institute of Technology is not able to spend a large amount of money on the purchase of an automated software development tool for the SDW. Furthermore, with the abundance of such tools in the public domain or under the propriety of the U.S. Air Force, AFIT should not have to purchase these component tools. Thus, the selection criteria that all SDW component tools must be available to AFIT free of cost is established. Under this criteria, there exist three manners in which components may be obtained for the SDW. Potential SDW components may be either public domain software, Air Force proprietary software, or available to AFIT on some type of academic loan arrangement.

Since, there is also a serious shortage of manpower to work on the development of the SDW, there is an additional criteria imposed on the selection of SDW components that deals with the availability of component options. The manpower constraints on the SDW development do not permit a great deal of time to be spent on the installation of SDW components. As a result, it is strongly suggested that all

SDW components be available in VAX-11/780 (the SDW target computer) compatible format. This eliminates the timely re-hosting of foreign software systems. However, this criteria may be overlooked in the event of a component option who's potential merit to the SDW outweighs the cost of re-hosting.

The final category of selection criteria for the SDW components is previously mentioned in section 3.5.2 of this document. This criteria deals with the relative importance of the component to the realization of the SDW. Each functional category of SDW components is given a measure of importance to the SDW in Figure 26 of Chapter 3. These measures of importance reflect the significance of members of the tool group to the establishment of an effective software engineering environment. In order to obtain the highest measure of importance, the tool groups must be required to provide the minimal capabilities common to all software development environments or must be needed to provide support to the pre-implementation stages of software development. More details on this set of criteria is provided in section 3.5.2.

In accordance with this just established set of criteria, potential SDW components must be able to satisfy the design specifications of the SDW Preliminary Design, available to AFIT free of charge and in a VAX-11/780

compaitable version, and of significance to the initial version of the SDW as measured by the criteria set forth in section 3.5.2. With this set of criteria defined, the component tools for Version 1.0 may be selected.

4.2.2 Selection of the SDW Components. The greatest constraint on the selection of SDW components is that imposed by the limitations of manpower and time. As a result of this constraint, only those design specifications for tools that are specified to be of highest priority to the initial realization of the SDW are satisfied by SDW component selections for Version 1.0. Each of these most significant design specifications are listed below with the specific tools that will be used to satisfy them.

Design Specification Deemed to be of Greatest Significance to the SDW -----	Specific SDW Component Selection -----
Compilers	VMS PASCAL VMS FORTRAN VMS BASIC VMS COBOL
Consistency Checkers	Requirements Engineering and Validation System Extended Requirements Engineering and Validation System CIDEF AIDES
Debuggers	VMS DEBUGGER
Functional Design Tools	CIDEF Interim AUTOIDEF

	AIDES
Graphics Editors	AFIT Graphics Editor SYSFLOW
Help Facility	Built into SDW Executive
Information-Oriented Design Tools	Interim AUTOIDEF
Linkers/Loaders	VMS Linker
Requirements Definition Tools	Requirements Engineering and Validation System Extended Requirements Engineering and Validation System CIDEF Interim AUTOIDEF
Simulators	Integrated Decision Support System (IDSS)
Teach Routines	Built into SDW Executive SDW User Manuals
Text Editors	VMS EDT Editor VMS SOS Editor
Word Processors	RUNOFF Text Processor

These particular software development tools are chosen for incorporation into the initial version of the SDW because they meet all of the requirements imposed upon the selection process by the previously stated criteria. All of these SDW component selections are available to AFIT from one of two sources. The first source is the existing VAX-11/780 VMS environment located in the AFIT Digital Engineering Laboratory. The second source of components is

the sponsoring Integrated Computer-Aided Manufacturing/Systems Engineering Methodologies Group. The tools available from these two sources are quite satisfactory for the accomplishment of the primary Preliminary Design Module Specifications. Thus, no other sources are required in order to complete the initial implementation of the SDW.

Each of the specific SDW component selections satisfies the functional requirements of it's design specification tool group in a distinct manner. Thus, the selection of each SDW component requires a certain degree of justification. The following paragraphs provide this justification for each of the design specification tool groups.

The Compilers. The theoretical analysis of software engineering environments, provided in Chapter 2 of this document, points out that such environments must be able to support a variety of programming languages in order to allow the software developer to choose a language that most effectively meets his needs. To this end, four distinct languages are provided compiler support in the SDW. Each of the languages has its own merits and disadvantages. Thus, the software developer is given a greater deal of flexibility in language selection. Only the four languages are given compiler support because they are the only

languages supported by the AFIT/Digital Engineering Laboratory (DEL) VAX-11/780.

Consistency Checkers. The term, consistency checkers, spans a large variety of software tools. Consistency checkers are available to support products of almost all of the stages of the software life-cycle. The consistency checkers selected for the SDW are all embedded within Requirements Definition and Design tools. They provide specific support for the analysis of the products of these particular tools. The Requirements Engineering and Validation System (REVS) and the Extended Requirements Engineering and Validation System (EREVS) are used to develop and analyze system (software system) requirements. They store these requirements within internal data bases and the consistency checkers are used to analyze these requirements and report any consistency or completeness anomalies. The CIDEF tool is used to produce and analyze requirements or designs. This tool uses a subset of the IDEF0 methodologies and explicit data item definitions to describe requirements and/or designs. The IDEF0 diagrams and the data item definitions are then analyzed by an internal consistency checker for completeness and consistency. The CIDEF tool also has the capability to automatically generate FORTRAN 77 code from the complete IDEF0 models and data item definitions. The AIDES tool is provided by Hughes Aircraft through the ICAM/SEM office.

AIDES is a structured design tool that is used for the development and analysis of structure charts. AIDES has two distinct components. The first is a structure chart editor used to develop the models and the second is a consistency checker used to analyze the consistency and complexity of the structured chart models.

The Debugger. The VAX-11/780 VMS environment possess a symbolic debugger facility that is essentially language-independent. This debugger is thus a very powerful tool for the dynamic analysis of software code. As a part of the VAX-11/780 VMS environment means, this debugger is already the property of AFIT/DEL and is more than powerful enough to satisfy the debug requirements of the SDW.

The Functional Design Tools. Each of the three functional design tools approaches the concept of automated and interactive design in a different manner. CIDEF uses a subset of the IDEF0 methodologies and explicit data item definitions to describe the design for a software system. CIDEF has very powerful analytical capabilities. Not only does the tool provide for the production and analysis of design, but there is also a capability to generate FORTRAN 77 code from the completed designs.

The Interim AUTOIDEF tool is a prototype of a more extensive tool to be released at a future date. The Interim AUTOIDEF tool supports the drafting of IDEF0, functional design models; IDEF1, information models; and IDEF2, dynamic models. The Interim AUTOIDEF tool only recognizes its products as drafted diagrams, whereas the later AUTOIDEF will provide more analytical capabilities.

The AIDES Structured Design tool is provided to AFIT on academic loan from the Hughes Aircraft Co. This tool supports the development of functional designs as realized in Structure Charts. The tool provides two specific capabilities. First, a Structure Chart editor is used to enter and draft the models, then, an analyzer is used to check the models for completeness and complexity. The AIDES tool is a very powerful facility that is undergoing continued development and enhancement.

The Graphics Editors. Two distinct graphics editors are planned for inclusion in the SDW. One is a result of current thesis investigation in the AFIT/DEL being done by Capt. Kevin Rose. The other is called "SYSFLOW". This graphics editor is being re-hosted for the VAX-11/780 by ASD/AD and should be available for inclusion in the SDW by Fall 1982. Both tools provide interactive graphics capabilities and the AFIT editor also supports the use of color graphics.



The Help Facility. Most of the component tools provide there own on-line and off-line help facilities. However, the SDW user requires higher level help information on what tool are supported by the SDW and how to access them. This information is provided in the help facilities of the SDW Executive and the supporting off-line documentation.

The Information-Oriented Design Tools. The only specific information-oriented design tool is the Interim AUTOIDEF. This tool provides the capability to draft IDEF1 information models. Interim AUTOIDEF is purely a drafting tool, no analytical capabilities are provided. The graphics editors may also be used to draft other types of information models, if necessary.

The Linkers/Loaders. The VAX-11/780 VMS environment provides a resident Linker/Loader that provides all the capabilities required by the SDW.

The Requirements Definition Tools. There are four specific tools used for requirements definitions. The Requirements Engineering and Validation System (REVS) and the Extended Requirements Engineering and Validation System (EREVS) provide for the textual and/or graphical displaying of requirements, the consistency checking of requirements, and the simulation of requirements. The difference in the two systems is that EREVS provides greater analytical capabilities and allows for the stating and analysis of

concurrent requirements. These tools use the Systems Specification Language (SSL) and the Requirements Specification Language (RSL) to textually describe requirements and a process flow diagram called an R\_Net to graphically portray the requirements.

The Interim AUTOIDEF tool is used to draft IDEF0 models of requirements. The CIDEF tool may also be used to portray the requirements in limited IDEF0 models and then do some analysis on these models.

The Simulators. The Integrated Decision Support System (IDSS) is used to develop and execute IDEF2 dynamic models. These models may be developed either graphically or textually with IDSS. The IDEF2 models are network-oriented simulation models, however, they may be expanded with FORTRAN 77 code to provide for discrete event modeling. The REVS and EREVS also have limited embedded simulation capabilities. To realize these capabilities, PASCAL code must be entered into the statement of requirements.

The Teach Routines. The teach facilities for the SDW are embedded into the Help Facility of the SDW and the corresponding off-line documentation.

The Text Editors. There are two generally recognized types of text editors in existence today. They are line-oriented editors that edit text a line at a time and

screen-oriented editors that edit text a screen at a time. The SDW uses both types of editors. Both types of editors can be found within the VAX-11/780 environment and thus these editors are selected for inclusion into the SDW. The SOS editor is the standard Digital Equipment Corporation (DEC) line-oriented editor. The other editor is the DEC EDT editor that is a very powerful, screen-oriented editor.

The Word Processors. Only one word processor is available on the VAX-11/780 at the present time. This word processor is the RUNOFF Text Processor. Although not a true word processor, the RUNOFF Text Processor does provide for the efficient product of textual material. RUNOFF is an adequate substitute for a true word processor until one becomes available on the VAX-11/780 VMS environment.

#### 4.3 Detailed Design of the SDW Executive.

4.3.1 A Recursive Application of the Software Life-Cycle. The SDW component tools provide the SDW with its developmental capabilities. However, the high level requirements for the SDW state that it must be an integrated environment. The type of integration referred to here requires that all of the component tools be accessible through a common unified interface. This requirement is satisfied by the SDW Executive. The SDW Executive (SDWE) is

essentially a sub-system of the SDW. The SDWE sub-system is the interface to and controller of the SDW component tools. Obviously, the SDWE is, itself, a major development project. In order to develop an effective SDWE, a recursive application of the software life-cycle is performed on the SDWE. In particular, the detailed requirements for the SDWE are defined, a preliminary design is then developed, and the algorithms for the SDWE modules are established.

#### 4.3.2 Detailed Requirements Definition for the SDWE.

The SDWE is required to fulfill two roles within the scope of the SDW. First, the SDWE must provide an efficient and usable interface to components of the environment and, second, it must be able to control the execution of each of these components. Currently, the number of SDW components, as defined in section 4.2.2, is relatively moderate. However, the SDW is a developing environment and many additional tools are expected to be incorporated into the environment in the future. In order to control each of these components, the SDWE organizes each of the SDW components into a functional group. These functional groups are defined and justified in section 4.3.3 dealing with the SDWE Preliminary Design.

The first step in the development life-cycle of the SDWE is the definition of the specific requirements for the SDWE. The objective of this stage of the SDWE development

is to define the exact capabilities that the SDWE must possess. Two distinct Software Engineering Methodologies are candidates for the definition of the SDWE requirements. They are the Data Flow Diagram technique and the IDEF0 (ICAM Definition technique). As discussed in section 2.4 of this document, the Data Flow Diagram technique defines requirements in terms of data flows and data transformations. The IDEF0 technique uses control flows, input data flows, output data flows, mechanisms, and functional activities to define the system requirements. The IDEF0 technique is essentially the same technique as the SADT(TM) technique described in section 2.2. As an interface and controller routine, the SDWE is not easily described in terms of data flows and data transformations. Instead, the use of control specifications and functional activities is more appropriate to the explicit definition of the SDWE requirements. As a result, the IDEF0 methodology is chosen to describe the SDWE requirements. (Refer to section 2.2 for an explanation of the IDEF0 methodology as it is identical to the Structured Analysis and Design Technique described there.)

The particular diagrams of the IDEF0 model for the SDW Executive are enumerated in the table below:

## SDW Executive Requirements Model

### Node Title

A-0 Utilize the Software Development Workbench  
A0 Utilize the Software Development Workbench  
A1 Initialize the SDW  
A4 Execute the User's Command  
A41 Provide a Functional Tool Group  
A42 Provide Help Facility  
A43 Access the Pre-Fabricated Software  
Description Data Base

The top level diagram of the SDWE requirements model is shown in Figure 27. In this model, the SDWE is viewed as providing the ability to utilize the SDW. The utilization of the SDW is controlled by User\_Commands and accepts other User\_Input in order to perform its software development function. The output of the SDW utilization are Software\_Development\_Products.

The Activity "Utilize the Software Development Workbench" is analyzed by the Figure 28. This figure illustrates the breakdown of the activity into component activities. The first of these activities is the "Initialize the SDW" activity. This activity uses

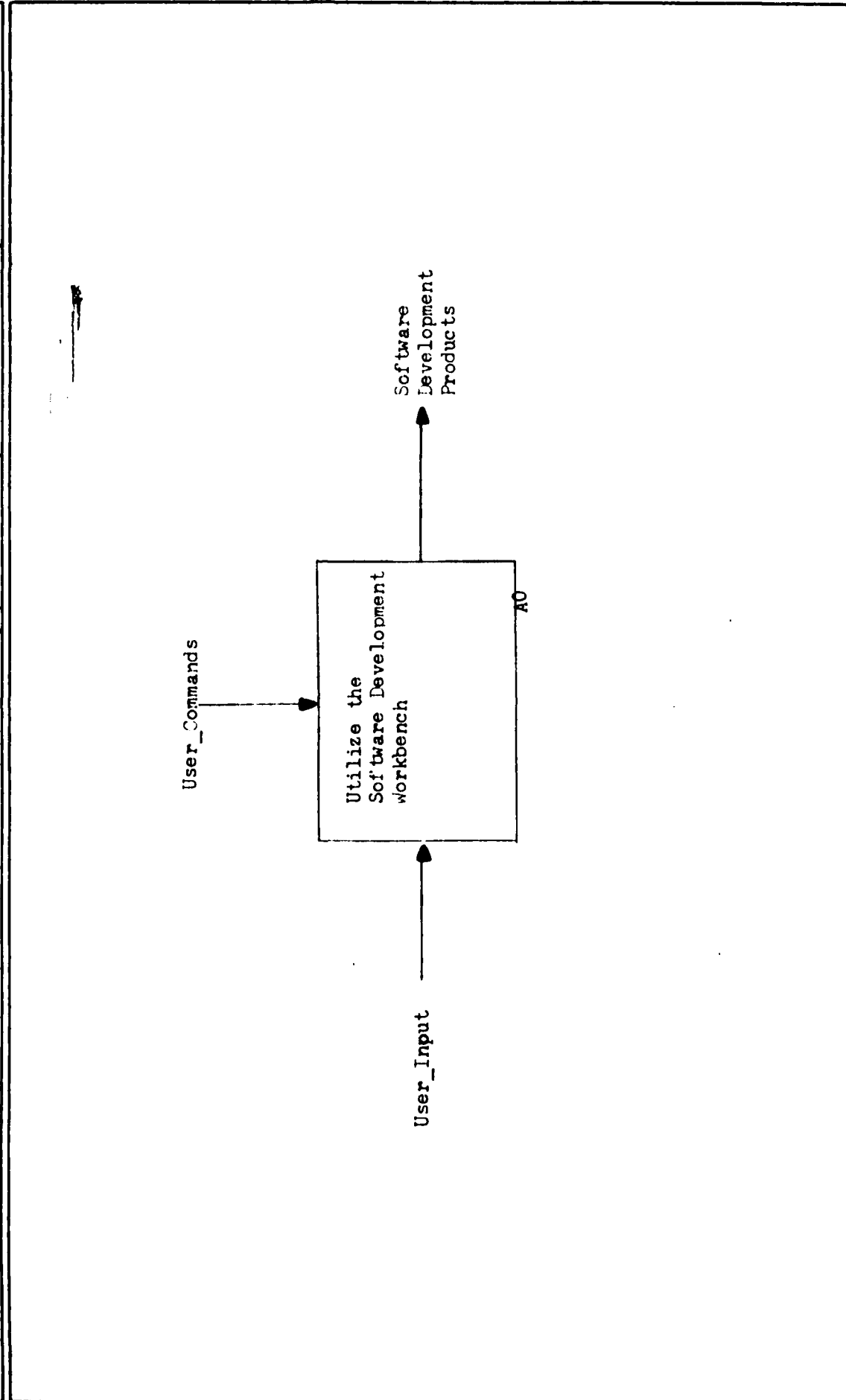
User\_Commands to control the initialization process and specify the enabling/disabling of the automatic menu facility. User\_Input is used to establish a data storage schema, whether it be a Project Data Base or the user's default directory. A detailed analysis of the "Initialize the SDW" activity is provided in Figure 29.

The second activity shown in Figure 28 is the "Provide a Set of Top Level Options". The function of this activity is to allow for only the SDW options that are appropriate to the top level module of the SDW. The manner in which the options are presented to the user is determined by the Auto\_Menu\_Flag and may use the Menu\_Files mechanism. This activity may also be re-initiated by the Return\_to\_Top\_Level\_Command. The output of the activity is a set of Top\_Level\_Options which may be executed in the next activity box in the figure.

This final activity, "Accept and Execute the User's Command", receives the User\_Command as a control item that is used to select and execute the Desired\_Option. Some of these options do require additional User\_Input. The resulting output of this activity is either a Conclusion\_Message, a Return\_to\_Top\_Level\_Command, or some type of Software\_Development\_Product. The "Execute the User's Command" activity is analyzed in greater detail in Figure 30.

ST252

USED AT:	AUTHOR: Lt. Steven Radfield	DATE: Oct 52	WORKING	READER	CONTEXT:
	PROJECT: SD&S Requirements	REV:	DRAFT		
			RECOMMENDED		
			PUBLICATION		
NOTES: 1 2 3 4 5 6 7 8 9 10					

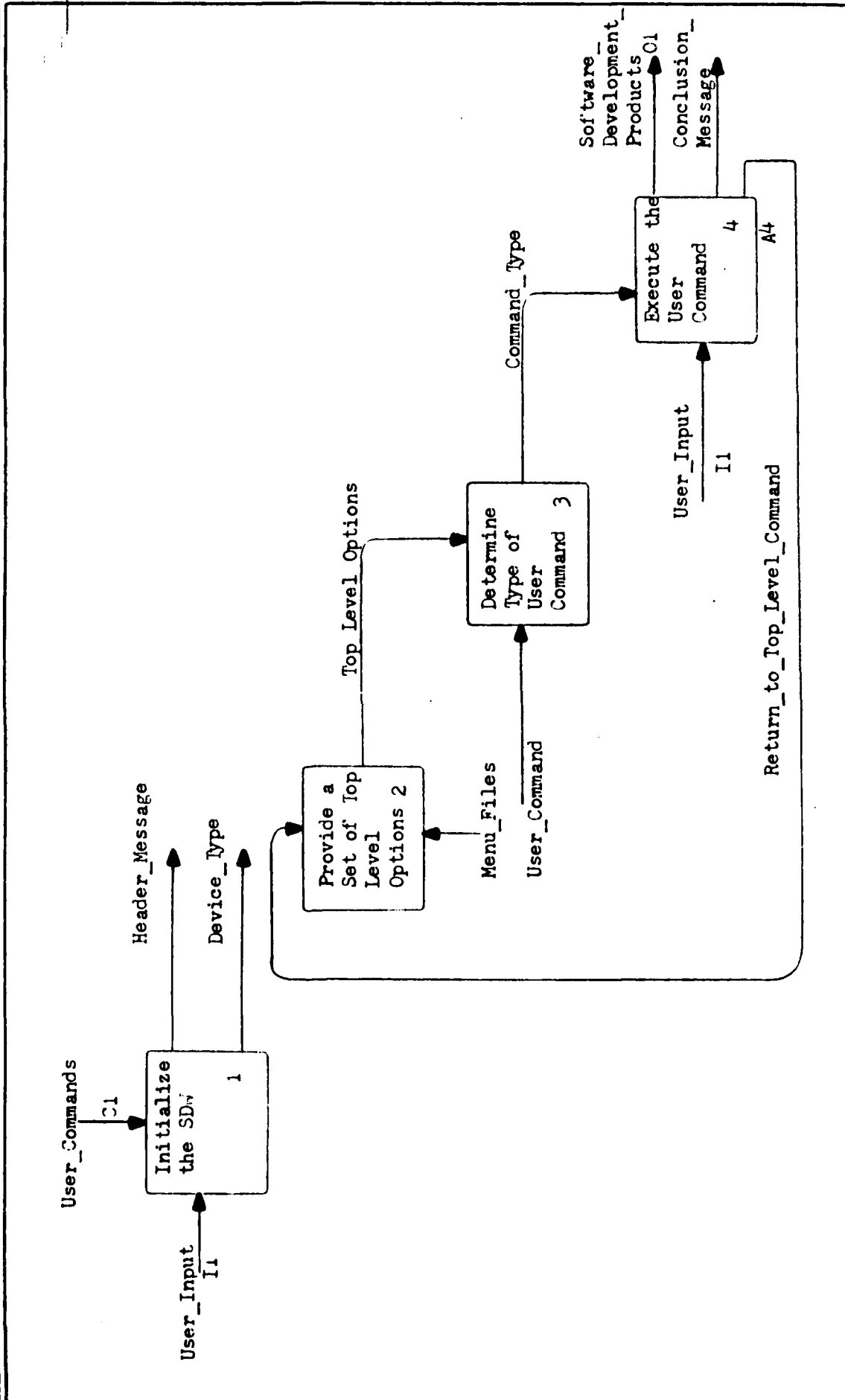


NODE: A-0	TITLE: Utilize the SD&S	NUMBER: SD&S1
-----------	-------------------------	---------------



SI 252

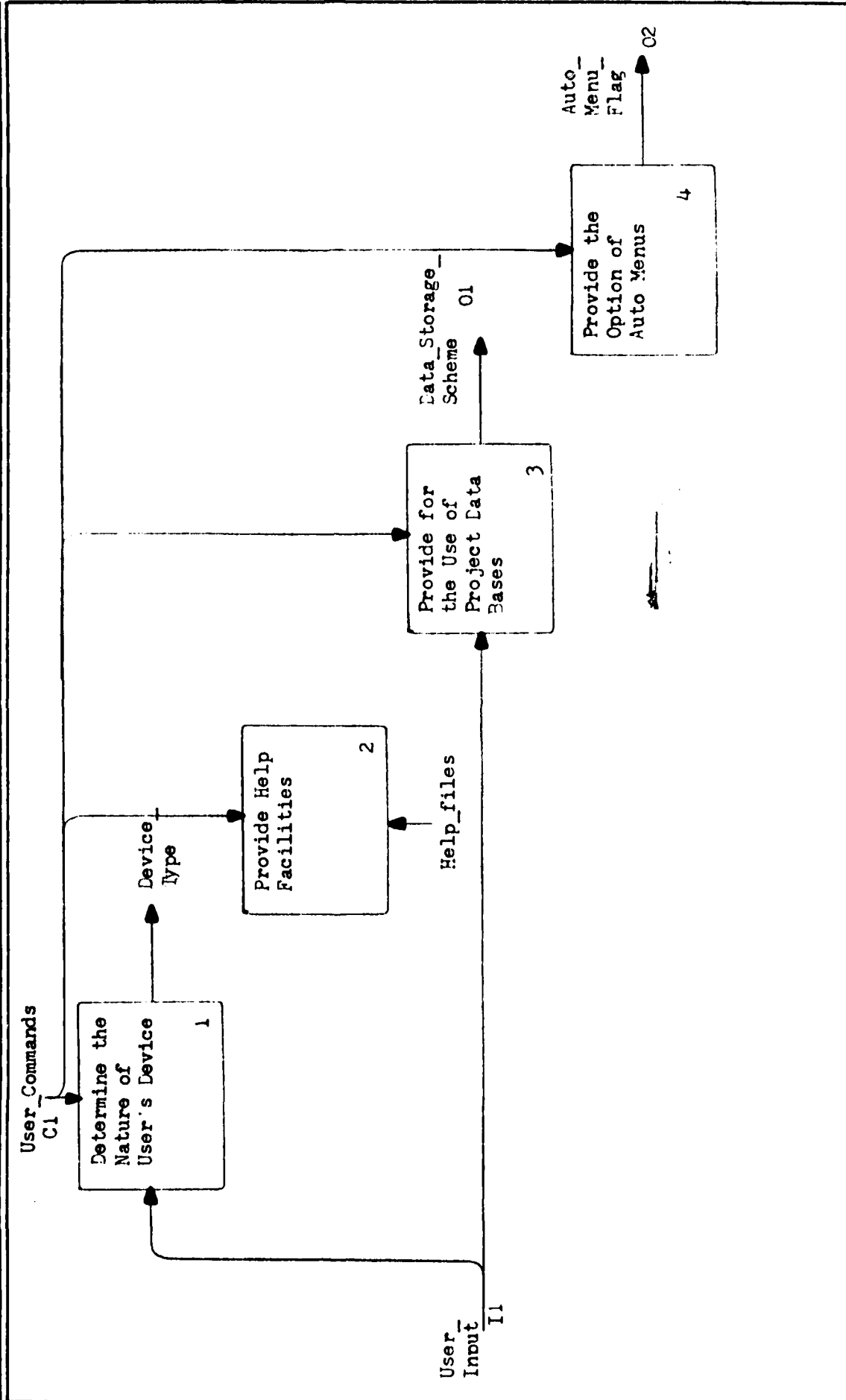
USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SDWE Requirements	REV:	DRAFT			
			RECOMMENDED			
			PUBLICATION			
NOTES: 1 2 3 4 5 6 7 8 9 10						A-0



NODE: A0	TITLE: Utilize the Software Development Workbench	NUMBER: 50002
----------	---	---------------

SI252

USED AT:	AUTHOR: Lt. Steven Radfield										DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SDME Requirements										REV:	DRAFT			<input type="checkbox"/>
	NOTES: 1 2 3 4 5 6 7 8 9 10											RECOMMENDED			<input type="checkbox"/>
												PUBLICATION			<input type="checkbox"/>
															40

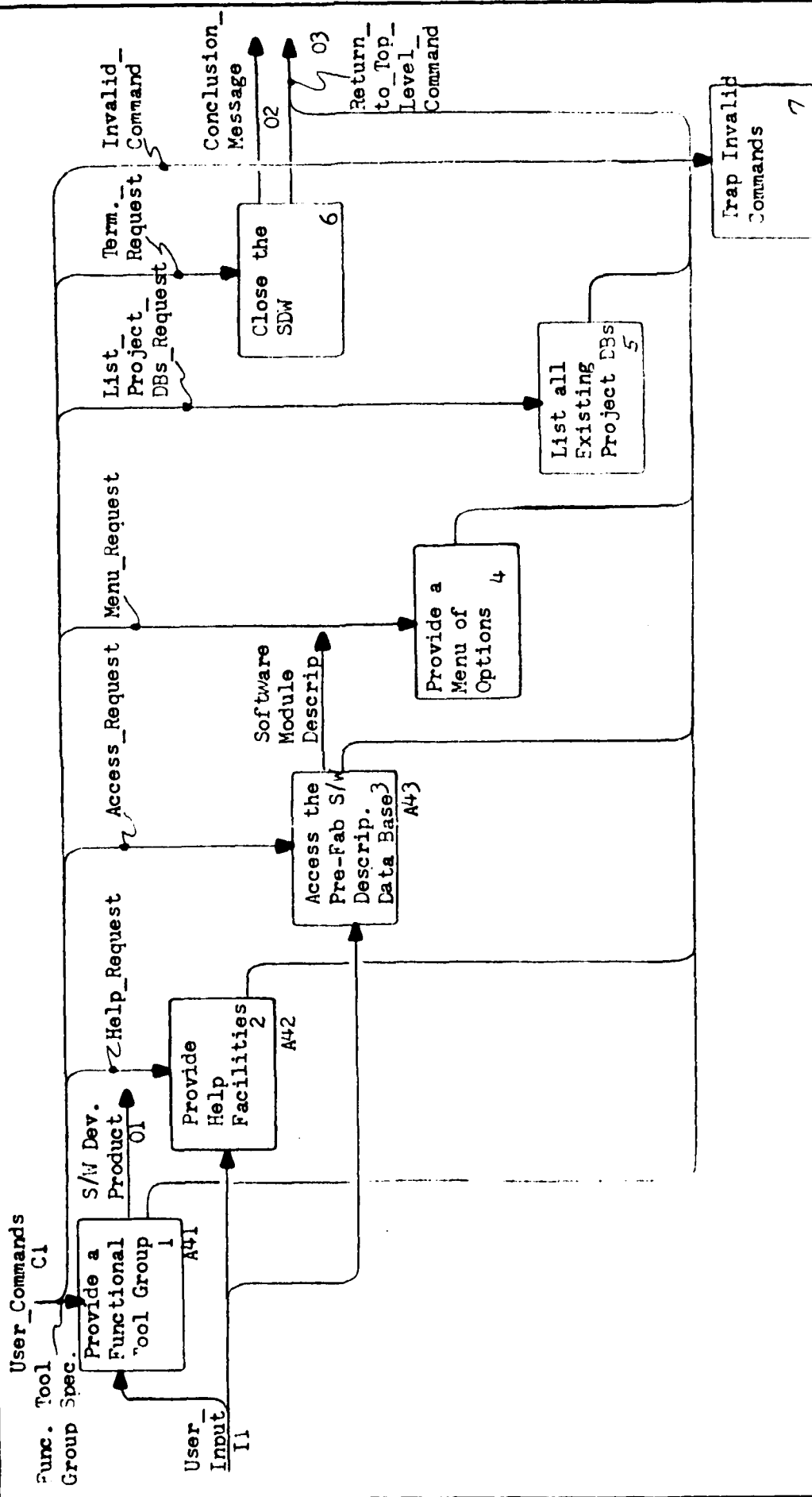


NODE: A.	TITLE: Initialize the SDME	NUMBER: SDME2
----------	----------------------------	---------------

The next diagram in the SDWE Requirements Definition Model is found as Figure 30. The User\_Command types are the Functional\_Tool\_Group\_Command that provides access to one of the fourteen functional tool groups, the Help\_Request that provides access to the SDW Help Facility, the Pre-Fab\_Software\_Description\_Data\_Base\_Access\_Request that provides access to that data base for either the addition or retrieval of a software module description; the Menu\_Request, that provides a display of the current menu options; the List\_Project\_Data\_Bases\_Request, that provides a display of all existing Project Data Bases; the Termination\_Request, that causes the graceful completion of the SDW session; a DCL\_Command, that is any of a set of host, monitor level commands that are also available to the SDW user; and then the Invalid\_Command, that must be trapped and recovered from by the SDWE. Each of these commands are used as control by a specific activity on this diagram. The "Provide a Functional Tool Group", "Provide Help Facilities", and "Access the Pre-Fab S/W Description DB" activities are all analyzed in greater detail in the following diagrams shown as Figures 31-33.

ST252

USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
PROJECT: S/W Requirements	REV:		DRAFT			
NOTES: 1 2 3 4 5 6 7 8 9 10			RECOMMENDED			
			PUBLICATION			A0



NODE: A4

TITLE: Execute the User's Command

NUMBER: S/W204

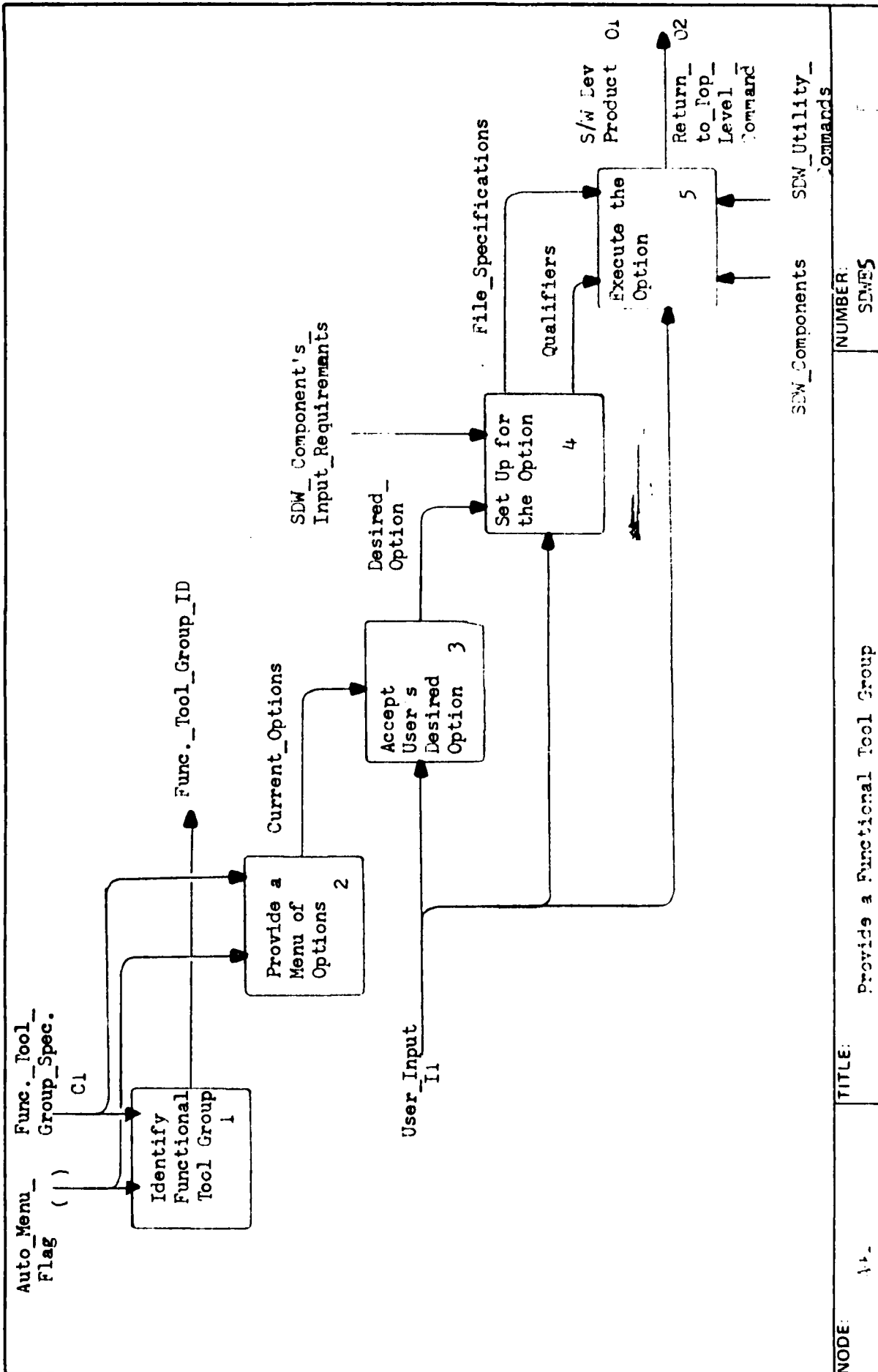
The "Provide A Functional Tool Group" activity illustrated in Figure 31 is very similar to the A0 diagram. The initial activity is to "Identify the Functional Tool Group". This is done with either a header message or a menu of options depending on the setting of the Auto\_Menu\_Flag. The SDWE functional group must then provide for the options required by that tool group. The Desired\_Option is then selected, the option set up for execution, and then executed.

The next diagram, "Provide Help Facilities", is illustrated in Figure 32. This diagram specifies what is required of the SDW Help Facility. In general, the SDW Help Facility must be able to provide either generalized help on the SDW, specialized help on any SDW component, and help on the DCL commands through the VMS Help Facility.

The final diagram of the SDWE Requirements Model is the "Access the Pre-Fab Software Description Data Base" diagram found in Figure 33. This diagram specifies that the SDWE must provide means to both add and retrieve descriptions of existing software modules to and from the Pre-Fab Software Description DB. Keywords are to be used to access these descriptions.

ST252

USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SDWE Requirements	REV:	DRAFT			<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			A4
			PUBLICATION			



NODE:

V1.

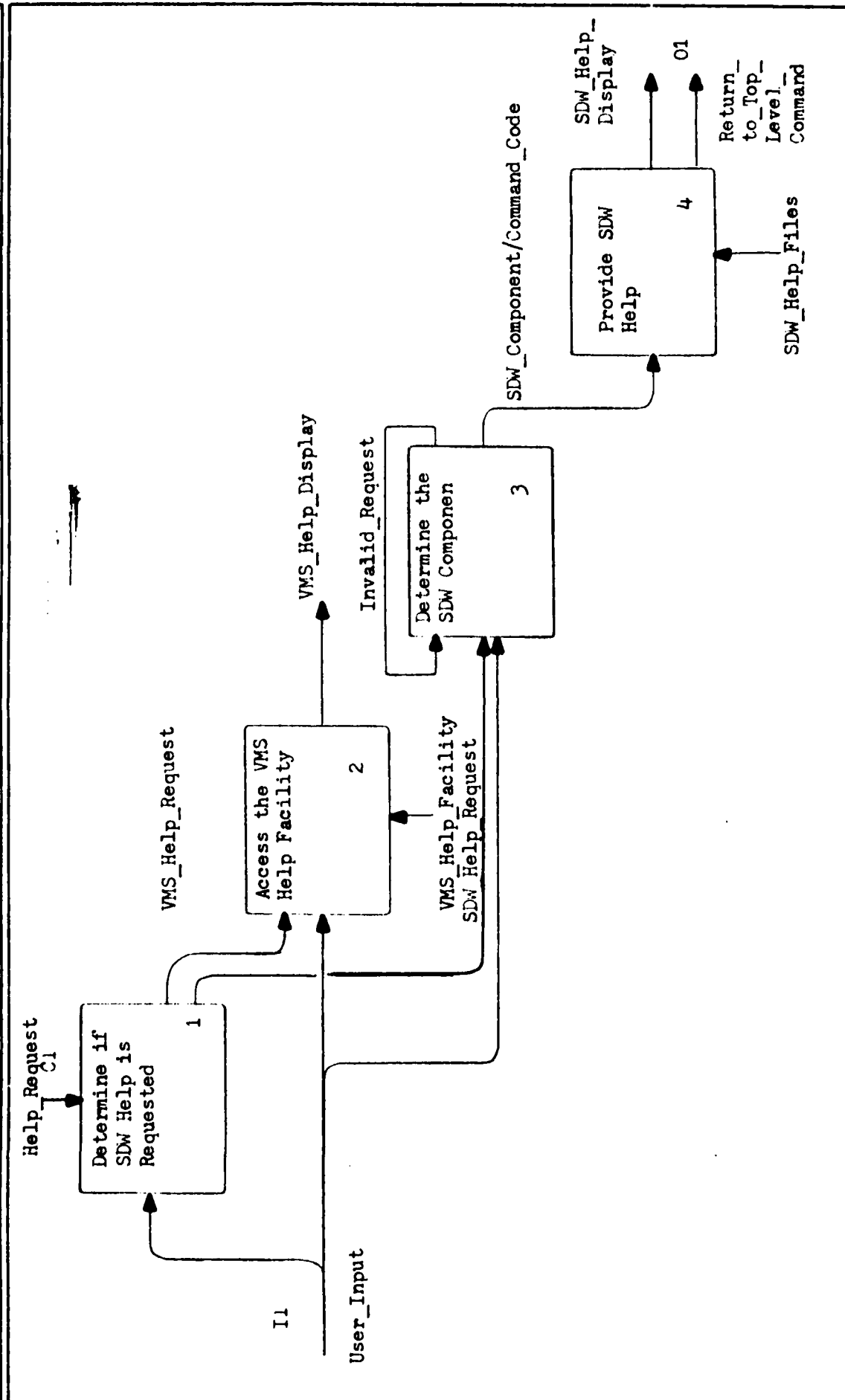
TITLE:

Provide a Functional Tool Group

NUMBER:

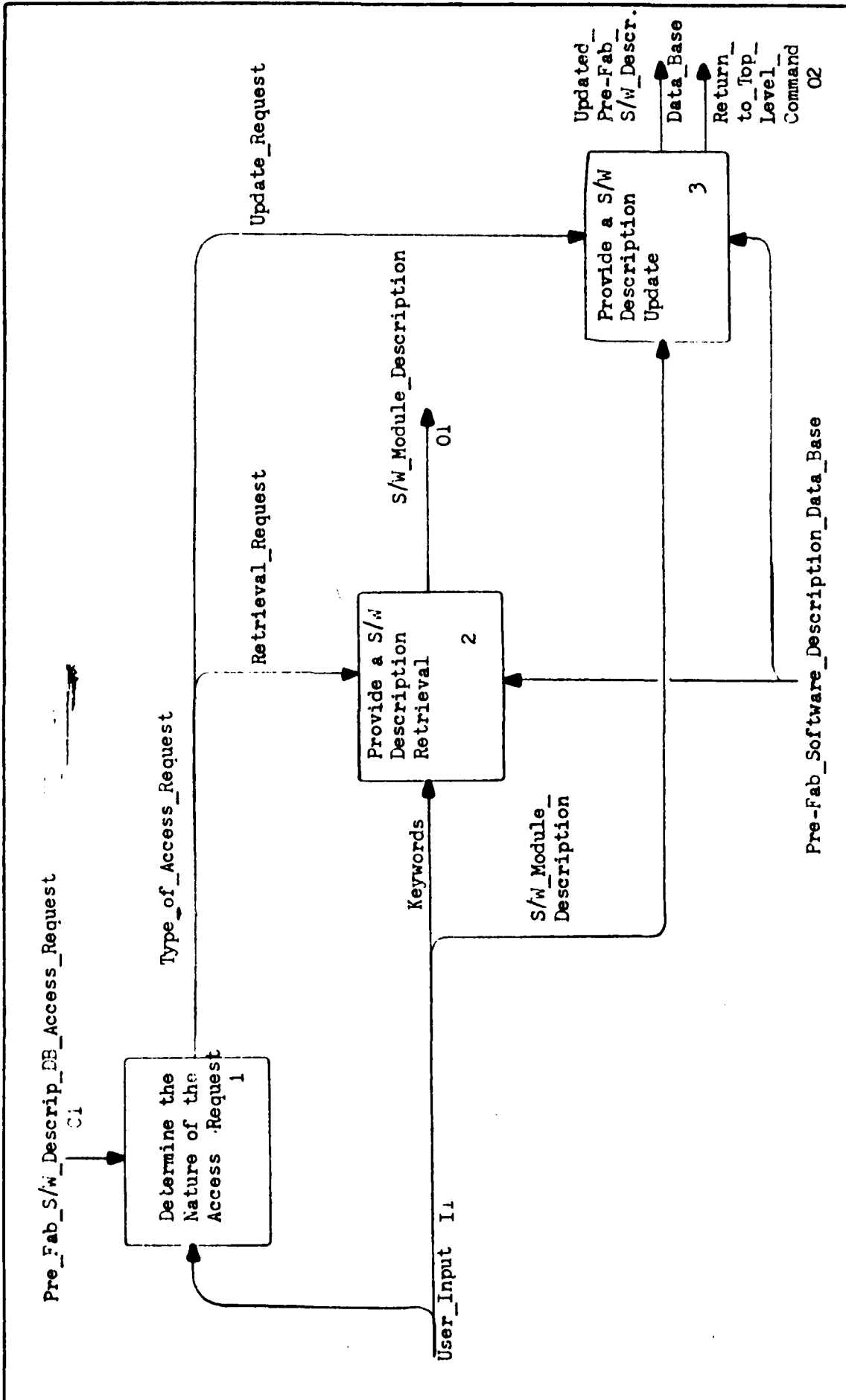
SDWES

USED AT:	AUTHOR: PROJECT:	DATE: REV:	WORKING DRAFT	READER	DATE	CONTEXT:
	Lt. Steven Hadfield	1 Oct 82				<input type="checkbox"/>
	SDWE Requirements					<input checked="" type="checkbox"/> <input type="checkbox"/>
	NOTES: 1 2 3 4 5 6 7 8 9 10					<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
			PUBLICATION			A4 <input type="checkbox"/>



SI252

USED AT:	AUTHOR: Lt. Steven Radfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SWE Requirements	REV:	DRAFT			<input type="checkbox"/>
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			<input type="checkbox"/>
			PUBLICATION			<input type="checkbox"/>
						44



NODE: A43	TITLE: Access the Pre-Fab Software Description Data Base	NUMBER: SDWE7
-----------	--	---------------



The SDWE Requirements Model just illustrated, demonstrates what capabilities must be provided in the SDWE in order for that SDW sub-system to perform its functions of controlling the SDW components and providing an efficient interface to those components. This statement of the detailed requirements of the SDWE is used for the initial development of the SDWE. Realizing that these stated requirements are subject to modifications resulting from experiences with the initial implementations of the SDWE, a post-implementation statement of the modified requirements for the SDWE is included as Appendix D.

It is also very important to establish a set of implementation language requirements for the SDWE. The set of requirements delineated below represents a minimal set of requirements for the SDWE implementation language.

#### Language Requirements for the SDWE

---

- 1- The language must be available on the target machine (VAX 11/780).
- 2- The language must have facilities for conditional branching.
- 3- The language must support modular design.
- 4- The language must provide input/output facilities for data, as well as, other information handling facilities.
- 5- The language must be able to control access to and the execution of the SDW component tools.

These requirements reflect the "essential" capabilities that a language must possess for use in the implementing of the SDWE. Other facilities, such as strict data typing and structured programming support, are also desired but not necessary.

4.3.3 Preliminary Design of the SDWE. The development of a preliminary design for the SDWE involves the establishment of a modular, top-down structure for the SDWE. This structure must contain modules that are specified to perform each of the previously stated SDWE requirements. In order to insure that the preliminary design satisfies the requirements, each design module of the preliminary design is mapped back to the specific requirements that it fulfills.

The SDWE requirements call for the SDW components to be assembled into functional groups. This is done in order to provide for the easy adding of new SDW components and to provide a more user comprehensible interface to the SDW component tools. These SDW Functional Tool Groups need to be specified prior to the development of the SDWE Preliminary Design because distinct modules are required to interface with and control each of these tool groups. The SDW Functional Tool Groups specified for the SDW are established to provide support to each of the generic tool groups specified in the preliminary design of the SDW stated

in Chapter 3. These tool groups are:

SDW Functional Tool Groups

- 1- Comparators
- 2- Compilers
- 3- Debuggers
- 4- Design Tools
- 5- Dynamic Analysis Tools
- 6- Editors
- 7- Graphics Editors
- 8- Linkers
- 9- Performance Monitors
- 10- Requirements Definition Tools
- 11- Simulation Tools
- 12- Static Analysis Tools
- 13- Test Case Generators
- 14- Word Processors

The only tool types not specified by a precise tool group are code generators and configuration managers. Code generators are mostly design tools with enhanced capabilities, so this type of tool is considered under the design tool's group. Configuration managers are not given a tool group because they should really be incorporated into the SDW Executive or the Project Data Bases.

These functional groups are derived from the SDW Preliminary Design Modules Specifications (Appendix C). Some of the design modules specified are grouped into a single functional group because the functions of the design modules specifications are logically realized as a single type of functional capability. For example, the design specifications referred to as Functional Design Tools, Information-Oriented Design Tools, and Consistency Checkers are often realized in single SDW components such as Interim AUTOIDEF. Thus, the generic term, "Design Tools", is used to refer to tools satisfying any of these design specifications.

There are two candidate methods available for expressing the Preliminary Design of the SDWE. They are the IBM Hierarchical Input Process Output (HIPO) technique and the Structure Charts technique. Both techniques support top-down structured design and specify the inputs and outputs to each module of the design. The Structure Chart technique distinguishes between data inputs and output and control inputs and outputs. The HIPO technique does not make that distinction, however, it does provide a more detailed and algorithmic definition of the module's function (the process). The Structure Chart technique simply uses a functional title for the module's process.

Either of the two techniques could effectively be used to define the SDWE Preliminary Design. The Structure Chart technique is chosen for three particular reasons. First, the Structure Chart does not define the algorithms for the design modules. These algorithms are the objective of the detailed design. The preliminary design's objective is to simply define the system's structure. By using the Structure Chart technique, later modifications to the algorithms of the SDWE need only require alteration of the detailed design. The Structure Chart technique more accurately satisfies the scope of the preliminary design stage as that stage is defined in this document (Section 1.3). The second reason is that the Structure Chart technique provides for the distinction between data and control parameters. This is important to the design of the SDWE because the SDWE must use a number of control parameters and flags in order to accomplish the flexibility required to handle different user devices as well as users with different levels of experience. The final reason for the use of the Structure Chart technique instead of the HIPO technique is that one of the selected SDW components (Hughes' AIDES) supports the automated and interactive development of Structure Chart models. There is no tool found to be available to AFIT for supporting the development of HIPO models.

With these tool groups specified, the Preliminary (or structural) Design of the SDWE may be developed. The preliminary design top level module specified for the SDWE is presented in Figures 34 and 35. This design module, called SDWEXE, provides the high level interface for the SDW user and controls all of the other modules that are used to control and interface to the various SDW components and SDW commands. In particular, the SDWEXE module must fulfill the SDWE requirements stated in SDWE Requirements Model diagrams A0 and A2, as well as boxes 4,6, and 7 of A4. The SDWEXE module calls the other modules of the SDWE to perform the remaining activities of diagram A4. Each of the SDWE Preliminary Design modules identified with an asterisk (\*) is a SDW Functional Tool Group Module that must satisfy the requirements of diagram A41 (Figure 31) for that particular tool group. The entire SDWE Preliminary Design is specified in Appendix E.

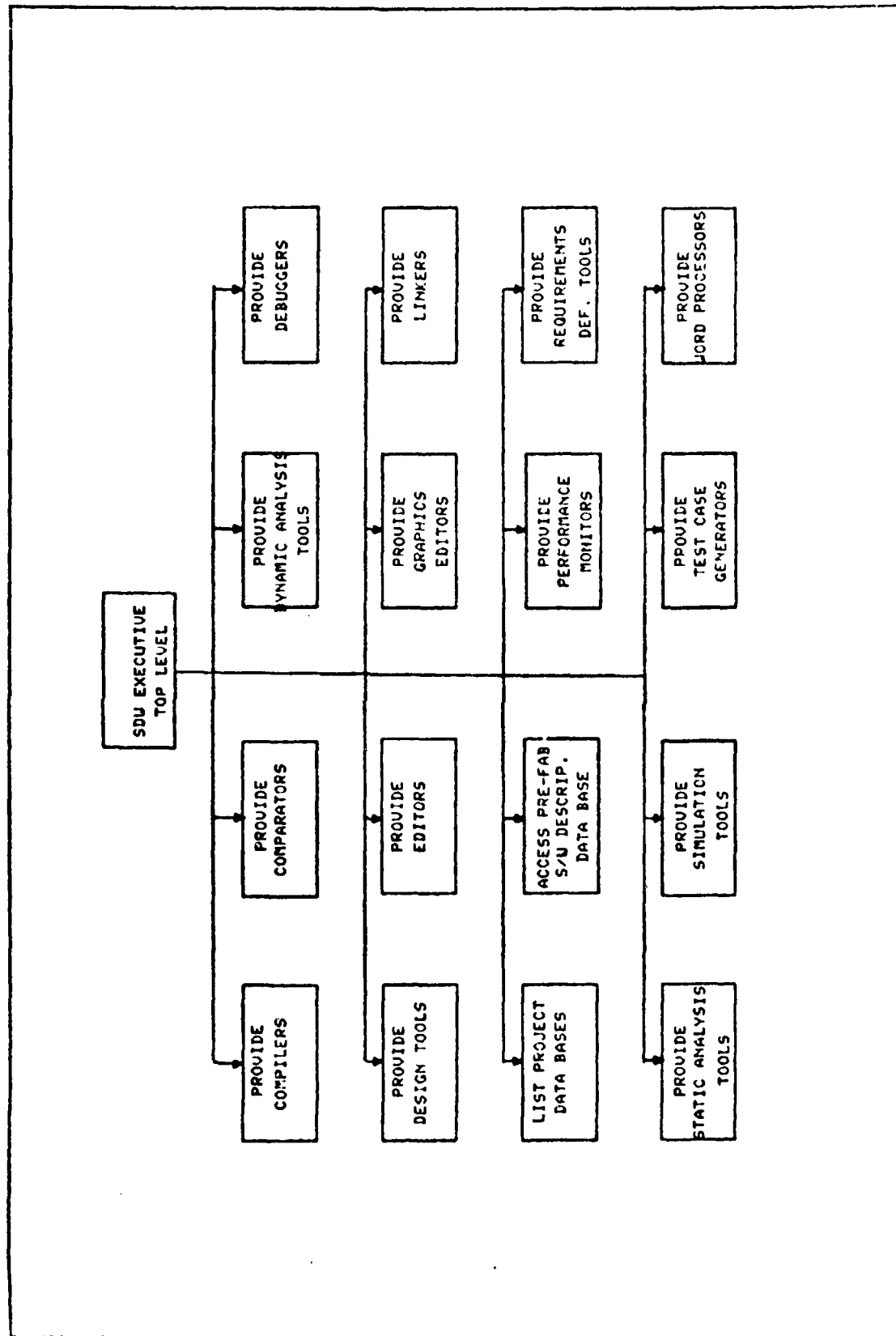


Figure 34: SDWE Preliminary Design Model

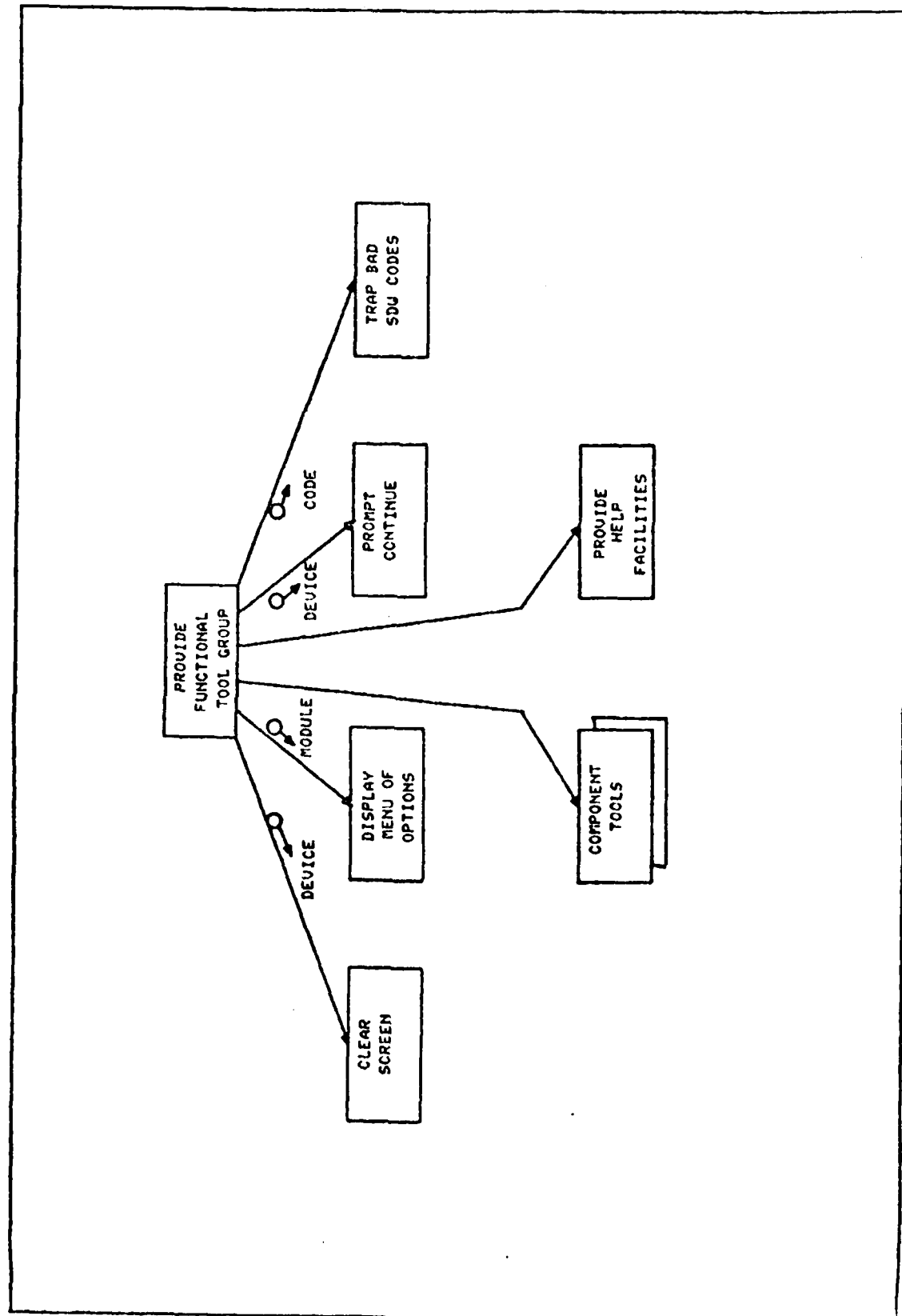


Figure 35: Functional Tool Group Structural Design



The other SDWE Preliminary Design modules that are subordinate to the SDWEXE module are used to fulfill the requirements defined by the other activity boxes of diagram A4. In particular, the Help Facility and SDW Help Facility modules are used to satisfy activity diagram A42. The List Project Data Bases design module fulfills box 5 of diagram A4 and the Access Pre-Fabricated Software Description Data Base module satisfies diagram A43. The Trap Bad Commands design module is used to satisfy box 7 of diagram A4.

Each of the SDWE Preliminary Design modules subordinate to the SDWEXE module are passed two control parameters. The first of these parameters is "Device" which contains the name of the user's device and is to be used for the exploiting of device specific feature by the modules and components. The second control parameter is "Prompt" which is the Auto\_Menu\_Flag specified in the SDW Requirements Model. This control parameter dictates whether the current menu of options is automatically displayed at every command prompt.

There is also a final preliminary design specification that is not explicitly shown in the SDW Preliminary Design Model. This design specification states that the SDWEXE module sets up the Data\_Storage\_Scheme and all project development data be stored according to the Data\_Storage\_Scheme. The Data\_Storage\_Scheme may be either

a Project Data Base or the user's default directory.

This SDWE Preliminary Design specifies the structure to be used in developing the SDWE. Like the previously stated detailed requirements for the SDWE, this preliminary design is subject to modifications that may be required by experiences with the initial SDWE implementation. The updated version of the SDWE Preliminary Design is included as Appendix E. The components of this level of design are simply titled boxes. The next and final step in the development of the SDWE sub-system is to use the SDWE Requirements Model and the SDWE Preliminary Design to develop a SDWE Algorithmic Design.

4.3.4 Algorithmic Design of the SDWE. Within the software life-cycle, the detailed design stage is characterized by the development of the design module algorithms. In order to better describe the objective of this stage, it is referred to as the Algorithmic Design stage in the rest of this document. The design modules for the SDWE are specified in the SDWE Preliminary Design model (Figure 34-35). Explicit algorithms for these design modules are developed using the SDWE Requirements model (Figure 27-33) for a reference.

The algorithms for the SDWE design modules are expressed in Structured English (Ref 90:48-49). The other Software Engineering Methodologies available for the specification of the algorithms are Decision Tables (Ref 90:49) and Decision Trees (Ref 90:49). Neither of these two options are very applicable to the development of the SDWE, whereas, the flexibility of Structured English makes it the most appropriate choice.

The subset of Structured English used to describe the SDWE algorithms uses a limited set of constructs, action verbs, data items, and other english words to formulate the algorithms in an easily understandable form. The Structured English constructs of the SDWE algorithms are the IF\_THEN, IF\_THEN\_ELSE, and REPEAT\_UNTIL. Structured English is a very useful tool for describing algorithms because of its understandability. As a result, the reader who is unfamiliar with Structured English should be able to pick up it's concept without much difficulty.

The highest level design module of the SDWE Preliminary Design model is the SDWEXE module. This module must satisfy the requirement specifications defined by diagrams/activity boxes A0, A1, A4, A41, A42, A43 of the SDWE Requirements model (refer to Figures 28-33). The algorithm for satisfying these requirements is detailed below:

### SDWEXE Algorithm

```
-----
(* Initialize the SDW *)

WRITE SDW_Header_Message
GET User_Device
GET Help_Request
IF Help_Request equals true THEN
    CALL SDW_Help_Facility
IF Project_DBs are in use
    GET Project_DB_Request
    IF Project_DB_Request is true THEN
        GET Project_DB_Name
        IF Project_DB_Name does not
        already exist THEN
            CREATE Project_DB_Name
        SETUP Project_DB_Name
GET Auto_Menu_Flag

(* Provide a set of Top Level Options *)

DEFINE Functional_Tool_Group_Codes
DEFINE Help_Command
DEFINE Menu_Command
DEFINE List_Project_DBs_Command
DEFINE Access_Pre-Fab_S/W_Descrip_DB
DEFINE Termination_Command

(* Accept and Execute the User Command *)

REPEAT_UNTIL User_Command equals Termination_Command
    IF Auto_Menu_Flag is true THEN
        DISPLAY Top_Level_Menu
    GET User_Command
    IF User_Command is invalid THEN
        CALL Trap_Bad_Commands
    EXECUTE User_Command
END_REPEAT_UNTIL
CLOSE Project_DB_Name
WRITE Conclusion_Message
```

Although each SDW Functional Tool Group controls and interfaces to a different set of SDW components, the manner in which each of the functional tool group modules perform their functions is very similar. As a result, a generic

algorithm is provided for these modules. This algorithm is presented below:

#### Functional Tool Group Module Algorithm

-----

```
REPEAT_UNTIL User_Command equals Return_Command
  IF Auto_Menu_Flag is true THEN
    DISPLAY Current_Menu
  ELSE (Auto_Menu_Flag is false)
    SO DISPLAY Functional_Tool_Group_ID
  DEFINE Help_Request, Menu_Request, Return_Command
  GET User_Command
  IF User_Command is invalid THEN
    CALL Trap_Bad_Commands
  IF User_Command requires parameters THEN
    GET Parameters
  EXECUTE User_Command
END_REPEAT_UNTIL
RETURN to SDWEXE
```

There are several other design modules in the SDWE Preliminary Design model that must have algorithms specified for them. They are the List Project DBs module, the Access Pre-Fab S/W Descrip. DB module, the Help Facility Module, the SDW Help Facility module, and the Trap Bad Commands module. The algorithms for these modules are defined below:

#### List Project DBs Algorithm

-----

```
IDENTIFY Project_DB_Names
WRITE Header_Message
WHILE more Project_DB_Names
  WRITE next Project_DB_Name
END_WHILE
RETURN to SDWEXE
```

Access the Pre-Fab S/W Descrip. DB Algorithm

-----

```
DEFINE Add_S/W_Descrip_Command
DEFINE Find_S/W_Descrip_Command
DEFINE Help_Command, Menu_Command, Return_Command
REPEAT_UNTIL User_Command equals Return_Command
    IF Auto_Menu_Flag is true THEN
        DISPLAY Current_Menu
    GET User_Command
    EXECUTE User_Command
END_REPEAT_UNTIL
RETURN to SDWEXE
```

Help Facility Algorithm

-----

```
GET Type_of_Help_Request
IF Type_of_Help_Request equals SDW_Help_Request THEN
    GET SDW_Component_Selection
    IF SDW_Component_Selection is "SDW" THEN
        CALL SDW_Help_Facility
    ELSE
        DISPLAY Appropriate_Help_File
ELSE (Type_of_Help_Request equals VMS_Help_Request)
    GET VMS_Selection
    CALL VMS_Help_Facility
RETURN to calling module
```

Trap Bad Commands Algorithm

-----

```
DISPLAY Bad_Code
EXPLAIN Bad_Code
```

### SDW\_Help\_Facility

PROVIDE Menu\_of\_General\_Help\_Options  
GET Help\_Option  
DISPLAY Requested\_Help\_File

These SDWE algorithms completely specify the SDWE in terms of detailed design. The algorithms avoid extremely low level specifications because those types of specifications are often implementation language dependent and the Algorithmic Design of the SDWE is meant to be language-independent. Furthermore, the use of the Project Data Bases is specified in very broad terms in these algorithms. The Project Data Bases may be designed and implemented in a variety of manners and their design and implementation should not be significant to the Algorithmic Design of the SDWE. Even though the design of the Project Data Bases is independent of the SDWE algorithms, it is still very fundamental to the overall SDW and, as such, is stated in the following section. The Algorithmic Design of the SDWE is also subject to modifications that may be required after the initial implementation of the SDWE. For this reason, the modified SDWE Algorithmic Design is included as Appendix F.

#### 4.4 Design of the Project Data Bases.

The Project Data Bases of the SDW are the means of storing the software development data and products produced with the SDW. As such, the Project Data Bases are a most fundamental sub-system of the SDW. The specific design of the Project Data Bases must be developed to fulfill the data storage requirements presented in the Requirements Definition chapter of this document. In particular, these requirements call for the integration of development data into a common data storage area (Section 2.3.6), the recording of relationships between the products of the different stages of the software development (Section 2.3.7), the means to easily access the development data for consistency and completeness checking (Section 2.3.13), and the ability to easily update the stored development data (Section 2.3.16). The concept of the integration of development data storage may actually be realized in two of the five levels of integration described in Section 3.4.6. The first of these two levels defines integration to be the storage of development data from the SDW into a single data area for each project supported by the SDW. The second, more demanding level of integration, requires the use of a Data Base Management System (DBMS) to store all of the development data and the relationships between the development data items.



Approach Taken in Designing the Project Data Bases.

The design of the Project Data Bases is presented in two stages. The first of these stages is the design for the immediate implementation of the Project Data Bases. The second stage presents a Project Data Base design suggested for ultimate implementation in order to fulfill all of the data storage requirements for the SDW. The first stage of the Project Data Base design is designed to fulfill the first level of integration of development data. This design consists of a single data storage area and is described with greater detail in the next paragraph. The second or ultimate design stage of the Project Data Bases attempts to fulfill all of the data storage requirements of the SDW. This design uses a DBMS that saves and preserves, not only the development data items, but also the relationships between these items. The design of the Project Data Bases is divided into these two stages because the design required to fulfill all of the SDW data storage requirements can not be fully realized until a full and complete set of SDW component tools is established. In order to at least accomplish some of the SDW data storage requirements, the first stage of the Project Data Base is presented.

The first stage of the Project Data Base design is developed to satisfy the first level of integration of the SDW development data. This level of integration requires that all of the development data for a specific software

development project be recorded in a common data storage area. However, each of the products of the different development stages are stored separately within this common data storage area, none of the relationships between the products are automatically recorded in this design. This design is established because it fulfills all of the data storage requirements that can be satisfied prior to the finalization of the SDW component set.

The second stage of the Project Data Bases design must satisfy all of the development data requirements for the SDW. This design must fulfill the second level of data storage integration. The design must preserve all of the development data items and the relationships between them, regardless of the development stage in which they were created. By fulfilling this second level of integration, the design also satisfies the requirement for traceability between development data items by means of the preserved relationships. Consistency and completeness checking of the development data must be supported by this design. The development data must be easily updateable also. Furthermore, the Project Data Base must be able to control the configurations of different versions of a software project. An example of this capability is found in the Source Code Control System (Ref 71,72).

In order to fulfill these requirements for development data storage, the second stage of the Project Data Bases design uses the concept of data schemas and a DBMS to automate the handling of the development data and the relationships between the data. A schema, in the sense of a DBMS, is an overall outline of the data items and the relationships between them. The schema is automatically enforced by the DBMS. A sub-schema is simply a subset of the schema that defines a logically related set of data items and relationships.

The second stage of the Project Data Base's design outlines a data schema for the Project Data Bases that must be handled by a DBMS. The outline of this data schema consists of the definition of the Project Data Bases sub-schemas and the requirements for each of these sub-schemas. The definitions and requirements for these sub-schemas are presented in generic terms because they can not be fully specified until all of the SDW components are satisfied.

The schema for the second stage Project Data Bases design calls for the use of seven sub-schemas. Six of these sub-schemas are used to store the development data that is realized as belonging to one of the following categories: requirements definition, preliminary design, detailed design, program code, testing activity data, and other

associated documentation, such as user manuals, maintenance guides, installation guides, etc. The seventh sub-schema is used to preserve the relationships between the elements of each of the other sub-schemas. The SDW users and the SDW components interface to the Project Data Bases through this seventh sub-schema. A model of this design is presented in Figure 36. This model is a formalization of a concept developed by the sponsor of this research investigation, Rick Mayer of the ICAM/Systems Engineering Methodologies Group.

By using the seven sub-schemas and a DBMS, the second stage Project Data Base design fulfills all of the requirements for integration of development data storage. Furthermore, the seventh sub-schema, that preserves the relationships between the products of the different development stages, provides for traceability between development data items and thus satisfies that requirement. The use of separate schemas for each stage of the software life-cycle allows consistency and completeness checking of stage products to be automated with data retrieval routines. (The Integration stage does not have a specified sub-schema because it does not involve the storage of extensive amounts of development data. It is replaced by a sub-schema for testing that holds the test data and test plans for the software system.) By using a DBMS, the updating of development data is easily accomplished with the automated

capabilities of the DBMS. Furthermore, the sub-schema defining relationships between the other sub-schemas allows updates to be traced through the other stages of the software development.

The detailed design and implementation of this second stage of the Project Data Bases design requires the establishment of a complete set of SDW components, an analysis of the data requirements for each of these components, and a development of the precise schema and sub-schemas required. After this stage of the Project Data Bases design is realized, interface routines must be developed to allow the SDW components to thrive off the Project Data Bases.

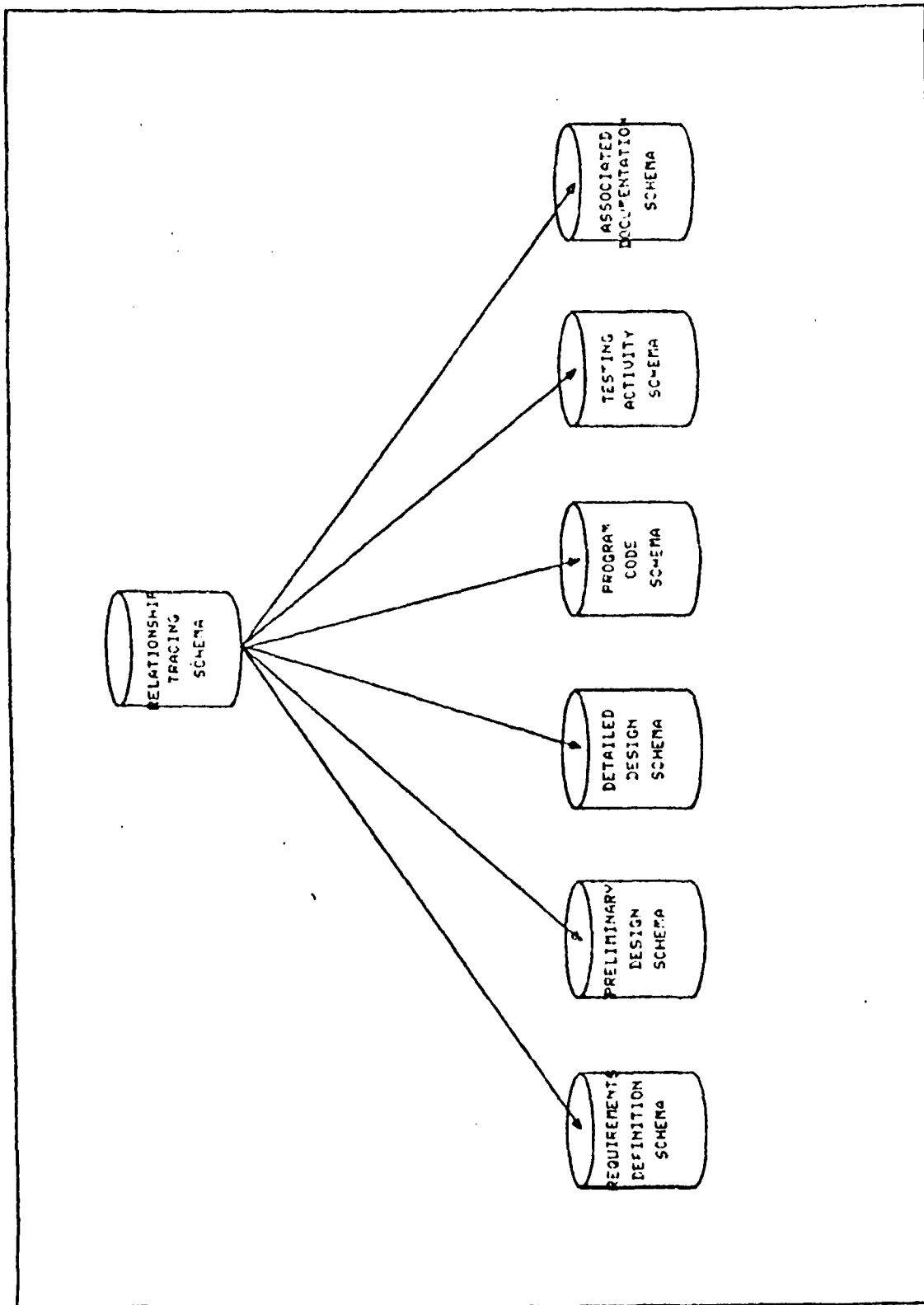


Figure 36: Project Data Base Design

#### 4.5 Summary.

As defined in section 1.3 of this document, the detailed design stage of the Software Life-Cycle involves the specification of the algorithms for the system under development. For this reason, the stage is often referred to as the algorithmic design stage. The development of complex systems often involves recursive applications of the Software Life-Cycle for component sub-systems. The detailed design of the SDW involves the specification of the SDW component tools as well as a complete development of an executive sub-system that must provide an integrated interface to the SDW components.

The selection of SDW component tools is highly significant to the effectiveness of the SDW as a software development environment because these tools provide all of the development facilities for the SDW. Only a limited number of tools are chosen for initial incorporation into the SDW because of two particular constraints on this phase of the SDW development effort. First, the target machine, the AFIT/DEL VAX 11/780, is a limited configuration that possess only two RK07 disk drives and no tape drive. Thus, the transfer, re-loading, and storage of potential SDW components is severely constrained. Second, the limited time involved in this phase of the SDW development effort allows only a limited number of potential component sources

to be investigated.

The development of the SDWE requires a recursive application of the Software Life-Cycle. The SDWE is a most significant sub-system of the SDW because it is the interface to and controller of the SDW components. The SDWE is what makes the SDW an integrated software development environment. In this chapter, the initial requirements definition and design of the SDWE is developed. The specifications are, however, subject to modification as determined during the later implementation and testing stages of the SDWE. As a result, the final specifications of the requirements and designs of the SDWE are stated in the models included as Appendices D, E, and F.

With the detailed design of the SDW thus specified, the initial implementation of the SDW may begin. The strategy utilized for implementation and the implementation specific decisions are described in the following chapter of this document.



AD-A124 872

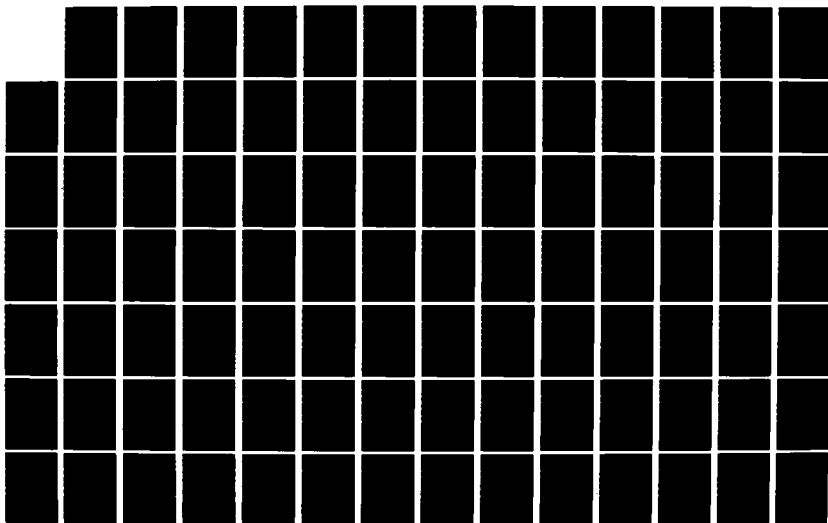
AN INTERACTIVE AND AUTOMATED SOFTWARE DEVELOPMENT  
ENVIRONMENT(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING S M MADFIELD DEC 82  
AFIT/GCS/EE/82D-17

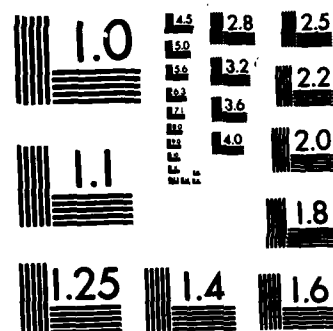
3/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

CHAPTER 5: THE IMPLEMENTATION STAGE

### 5.1 Introduction

The implementation stage of the software life-cycle is characterized by the conversion of the detailed design specification into executable code modules. This stage is not entirely distinct from the previous stage, detailed design, and the following stage, integration. Partial implementation is often accomplished prior to the development of a full detailed design specification. This is done in order to study the feasibility of certain design specifics prior to a full commitment to a specific design development. The use of incremental implementation is also useful in testing the developing system. Because of reduced complexity, this type of code testing tends to be very effective (Ref 90:208-225). The stage following the implementation stage is the integration stage. During this stage the individual sub-systems of the development effort are brought together into a single system. Sub-systems may be tied together prior to the complete implementation of other sub-systems. Thus, an overlap between the implementation and integration stages is evident.

The Software Development Workbench (SDW) is composed of the controller sub-system called the Software Development Workbench Executive (SDWE) and the SDW component tools. The specific objective of the implementation stage is to fully code and test the SDWE design specification of Appendices E

and F (Preliminary Design for the SDWE and Algorithmic Design of the SDWE, respectively). To this end, the implementation stage described in this chapter defines and justifies a choice of an implementation language, establishes an implementation strategy and discusses a complete set of implementation specific decisions. Also, the changes to SDW Version 1.1 are presented. Specific implementation of the SDWE utilizes a top-down approach. The overall modular structure is in Figure 37.

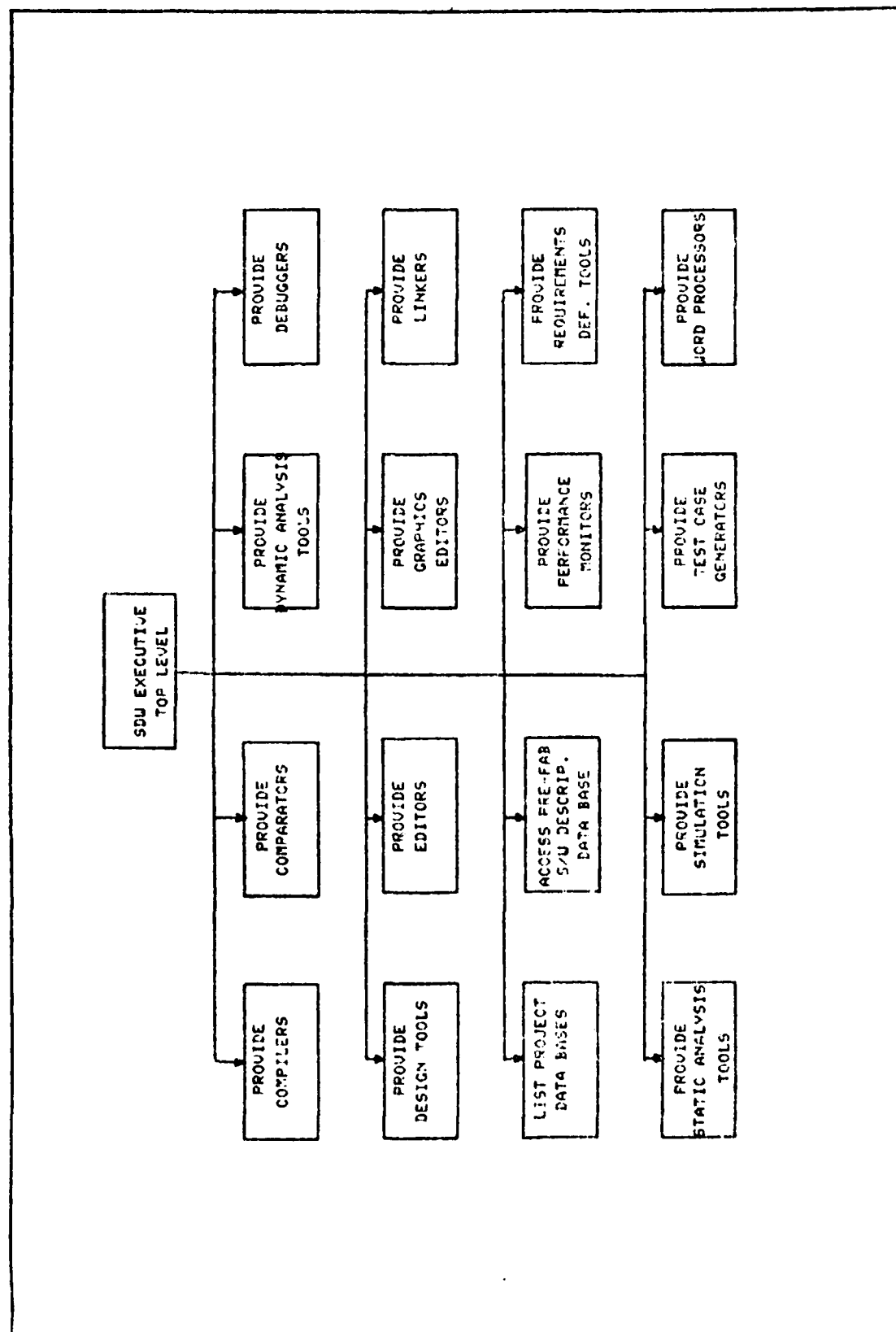


Figure 37: Preliminary Design Top Level

The Top Level module is coded and tested first. Then, the Provide Editors module is coded, tested, and integrated to the Top Level module and the SDW editor components. At this point, this initial implementation of the SDW can be used to code and test the rest of the SDWE.

## 5.2 The Choice of an Implementation Language for the SDWE

A fundamental decision that must be made before the implementation stage of development is the choice of an implementation language. A varied spectrum of computer programming languages are available to the contemporary programmer (Ref 60). Each language possesses its own individual features and limitations. A careful choice of a programming language can greatly reduce the time and effort involved in implementing and maintaining a particular software system.

Two categories of languages are studied and compared for potential use in the implementation of the SDWE. The first category of languages is quite large and very popular. This category is the Higher Order Languages (HOLs). Languages of this category are characterized by a set of powerful control flow structures and a variety of potential data types. The second category of languages under scrutiny for the implementation of the SDWE are command languages.

These are interpreted languages that are specifically designed for one type of operating system. Most command languages possess at least the primitive control structures of the HOL.

For the detailed analysis of these two categories of alternatives, available representatives of each category are chosen. There are two HOLs available on the target machine, FORTRAN and PASCAL. However, PASCAL possess much better information handling facilities than does FORTRAN (Ref 87). Since this application is primarily information handling oriented, PASCAL is the more favorable of the two options.

The Digital Equipment Corporation (DEC) Command Language (DCL) is selected to represent the command language category (Ref 27). DCL is the representative because it is the only command language available on the target machine.

Prior to the comparison of the two alternative languages, a set of language requirements for the SDWE are established as a measure of the analysis of the languages. These requirements are first stated in Section 4.3.2 which deals with the specific requirements for the SDWE. However, these language requirements are also delineated in the table below:



#### Language Requirements for the SDWE

- 1- The language must be available on the target machine.
- 2- The language must have facilities for conditional branching.
- 3- The language must support modular design.
- 4- The language must provide input/output facilities for data, as well as, other information handling facilities.
- 5- The language must be able to control access to and the execution of the SDW component tools.

These requirements reflect the "essential" capabilities that a language must possess for use in implementing the SDWE. Other facilities, such as structured programming support, are also desired but not necessary because the constructs of structured programming may be built with simple decision and branching statements.

The PASCAL language is a relatively new language, having been developed by Nicklaus Wirth in 1971. PASCAL possess exception information handling capabilities as provided by its varied and powerful data structures. PASCAL is a structured language meaning it supports a set of powerful control structures, in particular, the IF-THEN-ELSE, WHILE-DO, REPEAT-UNTIL, etc. These control structures are the essence of structured programming

support. PASCAL is also a relatively fast and efficient language to run because it is translated into machine code with a compiler. The major disadvantage of PASCAL in terms of the SDWE is that it can not interface to the command language routines because of the architecture of the VMS operating system. Thus, PASCAL may not be used to run the SDW component tools.

On the other hand, the DCL language may easily interface to the command routines of the SDW component tools since they are also implemented in DCL. There are, however, several disadvantages to using DCL for the SDWE. DCL possess only two data types, integers and character strings. The control constructs provided by DCL are very limited, consisting only of decision statements and branching statements. DCL is an interpreted language meaning that the code is translated and executed one line at a time. Thus DCL programs execute significantly slower than comparable HOL programs that are converted to machine language by compilers.

Although there are several disadvantages to using DCL for the SDWE implementation, these disadvantages are not stifling. The two data types provided by DCL are the only ones required for the SDWE. While the control constructs are limited, they are sufficient to create whatever other constructs are required. Furthermore, there are several

functions available for handling character strings. Even though the DCL language is slow, the SDWE spends most of its execution time waiting for user responses and the code executed between these I/O activities is not extensive. Thus, the slower execution time would not be significantly noticed by the SDW user. In addition, DCL provides means to trap interrupts whether they be initiated by the user or an error condition.

The preceding discussion reveals that DCL, while by no means "optimal" because of its slow execution speed and poor control structures, is the only available alternative that meets all of the previously stated language requirements for the SDWE (section 5.2). PASCAL is not sufficient because it lacks facilities to interface with the DCL routines that drive the SDW component tools. DCL is thus chosen as the primary implementation language for the SDWE. However, PASCAL may be called by DCL, so PASCAL is chosen as an auxillary implementation language to be used for some utility SDWE modules.

### 5.3 The SDW Implementation Strategy

The implementation stage described in this chapter for the Software Development Workbench (SDW) deals strictly with the implementation of the Software Development Workbench

Executive (SDWE). The SDWE is a sub-system of the SDW that is used as a primary interface to the SDW environment and a controller for that environment (section 3.3, Figure 22). The implementation of the SDWE is a sub-system implementation of the SDW. Concurrent to the implementation of the SDWE, the SDW component tools are being loaded on the target machine, the AFIT/DEL VAX-11/780. The SDWE is actually interfaced to the other SDW sub-systems, the SDW component tools, during the next stage, integration. This process is discussed in Chapter 6.

The implementation of the SDWE requires the coding of the design module specifications defined in section 4.3.3. An incremental approach to this coding is taken. This approach calls for the initial implementation of the top level module with dummy modules or stubs for all called routines. These dummy modules simply report that the module was properly called and then return control to the top level module. After the implementation of the Top Level module, the Provide Editors module is coded. With this module implemented, the SDW may be used to aid in the implementation of the rest of SDWE. This approach allows the incremental testing of the SDW implementation. Furthermore, by using the SDW to develop the SDW, the human interface features of the SDW algorithms are tested and possibly refined.

#### 5.4 SDWE Implementation Specifics

The implementation of the SDWE is accomplished by the incremental coding and testing of the algorithms for the SDWE that are established in section 4.3.3. During this implementation, a number of specific decisions are made to define how the requirements of the SDWE are to be met. This section reports and justifies the specific decisions that are made during the initial implementation of the SDWE.

Prior to the coding of the first SDWE module, a set of command and file conventions are established for the SDWE. The command conventions require that each SDW-specific command be a unique, two-letter code. The length of two letters for the commands is chosen because it provides many alphabetic combinations (26 squared or 676 combinations). Furthermore, the two letters provide a reasonable ability to describe the meaning of the commands without being excessively long to type in on the keyboard. The requirement for uniqueness of these codes is imposed as a result of the established file conventions. Each SDW-specific command must have an associated help message that is accessible from the SDW Help Facility. A separate file is used for each help message, thus allowing these help messages to be displayed with the "TYPE <file\_name>" command. The naming of these help files consists of the two letter code for the command, say "xx", followed by the word

"help". Thus, if the code for the command is specified in a variable entitled "OPTION", the help file is easily accessed with the following DCL command:

```
TYPE 'OPTION'HELP.MEM
```

The single quotes around the command variable option mean that the value of that variable is used in the command.

Those SDW-specific commands used to access the functional tool groups also require text files that state the menu of options within that tool group and identify the tool group. Files used for these purposes are named, "xx"MENU.MEM and "xx"ID.MEM respectively.

The implementation of the top level SDWE module, SDWEXE, is initiated with the defining of data and control variables that are used by all of the SDWE modules. These variables are stated and defined as follows:

### SDWE Data and Control Variables

---

Name -----	Type -----	Definition -----
device	string	The name of the user's CRT device
module	string	The ID for the current SDWE module
nopdbs	boolean	Enable/disenables the use of Project Data Bases
option	string	The current command option
pdb	boolean	Specifies if a Project Data Base is currently in use
project	string	Name of the Project Data Base in use
prompt	boolean	The Auto_Menu_Flag
qualifier	string	List of qualifiers for a particular SDW command
ret	string	Dummy variable used for accepting user input to continue
save_dir	string	Name of the user's original default directory

With the data and control variable specified, the actual coding of the top level SDWE module proceeds. The execution of the SDW is initiated with a header message that confirms the entrance into the environment and identifies the version number and authors. Then, a few queries are used to determine the type of CRT device in use, whether a Project Data Base is to be used, and if the user requires immediate help. These queries fulfill the algorithmic design of the SDWEXE module (Section 4.3.4). Then, a menu of the top level command options is presented with a prompt for user input. The prompt for user input consists of the SDW-specific code that was used to access the current SDWE level followed by a greater-than character and a colon as shown below:

XX> :

The two letter code is required to identify the current level of the SDW in the event that the auto menu facility is disenabled. The greater than character is used because it is a convention common to many VAX-resident packages, such as the resident debugger and the SYSGEN routine. The colon is an unavoidable feature of the DCL "INQUIRE" command that is used for reading data into a DCL program (Ref 27:235-237).



With the initial implementation of the SDWE top level module completed, the module to control the SDW editors is coded to enable the use of the SDW component editors to aide in the further development of the SDWE. Using the SDW in this fashion enables the critical review of the SDW algorithms prior to complete implementation.

One modification, made evident while using the SDW to develop the SDWE, is the need to have the screen cleared between distinct operations of the SDW. A primitive function, named "CLEARSCRN", is used to pass the necessary control characters to the user's CRT device to perform the clear screen function. The "DEVICE" variable is used as the control parameter for determining the proper control character string. The Tektronix 4014 and 4016 devices require a Form-Feed character. DCL lacks facilities for passing such a character, so a special PASCAL program, "CLEARTKTX", is used when these Tektronix devices are specified by the DEVICE variable.

The "CLEARSCRN" and a module called "CONTINUE" are the only device specific modules of the SDWE. The "CONTINUE" module is used to query for user permission to clear the display screen. This capability is important because it allows the user to view the entire display and clear the display.

When executing a DCL command program, such as the SDWE, the operating system expects all commands to be taken from the mass storage device on which the program is located. As a result, SDW components, such as editors, that require user input, are not executable. In order to alleviate this problem, the following command is used to flag the operating system to expect commands from the user's CRT device for the execution of the next component:

```
ASSIGN/USER_MODE SYS$COMMAND: SYS$INPUT
```

By including this command statement just prior to the execution of every SDW command option, any SDW component or any VMS component may be executed from the SDW input prompts. The only exception is that the entire command must be entered on a single line. This is a restriction of the VAX/VMS operating system.

One of the primary requirements for both the SDW and the SDWE is the inclusion of a Help Facility (sections 2.3.8 and 3.4.8). Since the SDWE is capable of executing both SDW-specific and VMS commands, the SDWE Help Facility must provide help capabilities for each type of commands. To this end, the SDWE Help Facility uses three components that are selectively accessed through the SDW help request command. One component is a general help facility for the SDW as a software development environment. The next

component provides help on selected SDW specific command codes. The final component is the resident VMS Help Facility used to provide help on VMS commands.

The SDWE is also required to trap invalid command code specifications (Section 4.3.2, Figure 30). The VMS operating system automatically traps invalid commands with warning messages. VMS also provides means to execute specific commands upon the occurrence of such a warning. This capability is provided with the ON WARNING THEN <command> facility (Ref 27:329-330). Within the SDWE, the occurrence of an invalid SDW command causes the "BADCODE" module to be executed. This module reports an error message to the user's terminal. The error message includes the invalid command. Control is then returned back to the calling module.

These are the implementation specifics resulting from the initial implementation of the SDWE. However, a number of experiences with this implementation, Version 1.0, demonstrated the need for an updated version of the SDWE. These experiences and the resulting modifications requiring the update to Version 1.1 are described in the next section of this chapter.

### 5.5 SDWE Update to Version 1.1

The driving forces behind the update of the SDWE to Version 1.1 are an operating system update from VMS Version 2.7 to VMS Version 3.0 and a critical review of the SDWE by the advisor for this thesis investigation, Dr. Gary Lamont. The result of this Version update is a more user-friendly and user-safe interface for the SDW.

The VMS operating system update from Version 2.7 to 3.0 requires only minor modification to the existing SDWE code. Version 3.0 of VMS uses a self-prompting hierarchical structure. Thus, the SDWE Help Facility is no longer required to prompt the user for specifics, such as commands and qualifiers. The SDWE Help Facility simply calls up the VMS Help Facility and lets that facility provide specific help on the VMS features. The second modification to the SDWE code deals with the facility for trapping erroneous command inputs. This is done with the following line of code Version 1.0:

```
ON WARNING THEN @SDW$DISK:[SDW]BADCODE
```

However, under Version 3.0 of VMS the explicit call to the Badcode module is not permitted. So the synonym, "TRAP", is defined to replace the call to Badcode as shown below:

```
TRAP :== @SDW$DISK:[SDW]BADCODE
      :
      :
      :
ON WARNING THEN TRAP
```

No VMS documentation is available that explains this peculiarity of Version 3.0!

The first set of SDWE modifications deal with the initiation procedures for the SDW. The initial header message is expanded to include, not only the system's name and version number, but also the names, addresses, and phones numbers of the system developers and copyright protection information for the SDWE.

All of the initial queries used to set up Project Data Bases, set the Auto\_Menu\_Flag, and provide initial help information are replaced by a new SDW module referred to as the SDW Utility Functions. The Device specification query still follows the header message. The initial default user CRT device is a VT100. After the Device query, a default device spec of a VT52 is used if no other device type is specified. All queries used in the SDWE check the user's response against the valid options and reprompt if an invalid response is detected. If the response range is not limited, as with the potential Project Data Base names, the

user's response is echoed for user validation.

The SDW Utility Functions provide the SDW user with a greater flexibility because of the ability to alter the Auto\_Menu\_Flag, the Data\_Storage\_Scheme, or the Device\_Specification at any time during the operation of the SDW.

The Auto\_Menu\_Flag is also used to provide automatic prompting of qualifier options for SDW\_specific commands that may use qualifiers. The displays of these options are stored in files whose names begin with the two-letter code for the SDW\_specific command and end with "PARMTS.MEM". This file name stands for the parameters that are available to qualify the particular SDW command.

Prior to exiting the SDW a number of concluding activities must take place. Under SDWE Version 1.0, the user could enter the DCL "EXIT" command that would cause the immediate exit of the SDW without the concluding activities taking place. To remedy this problem, the "EXIT" command is re-defined to a branch instruction that causes the "graceful" exiting of the SDW. A "graceful" exit requires that all the synonyms specific to the SDW be deassigned prior to termination of the SDW session.

User interrupts, CTRL/Ys and CTRL/Cs, as well as, error interrupts would also cause immediate exiting of the SDW without the concluding activities being performed. The ON-THEN DCL facility is used to trap these interrupts. This trapping involves the use of a graceful exit upon the occurrence of any of these interrupts.

The final modification of the SDWE for Version 1.1 deals with the scope of the SDW command codes. Only the command codes presented in the current menu of options should be executable from that level. DCL provides two types of scope specification for its synonyms, local and global (Ref 27:4-7). Version 1.0 defined all SDW command codes as local synonyms with the understanding that they would only be visible within the command module in which they are defined. This is an erroneous assumption! Once a command code is defined, it is visible until it is re-defined or deleted, or until the DCL command program is terminated. Thus, came the necessity to establish mechanisms to define and limit the visibility of the SDW command codes. The mechanism, used to achieve these visibility objectives, defines all current command code options globally upon entrance into a particular module and then deletes the synonyms when control is passed on from that module. The "ASSIGNSYM" module is used to define the proper command codes. The "MODULE" variable indicates the current command module to allow the proper synonyms to be

defined. When control is passed on from that module, the "DELSYMBOL" module deletes these command codes. Testing of this mechanism reveals that the required limiting of the SDW commands' visibility is achieved.

The updated Version 1.1 of the SDWE provides a more user-friendly interface to the SDW. The finalization of Version 1.1 marks the complete implementation of a major sub-system of the SDW. The next step is to integrate the other SDW components to the SDWE in order to realize the SDW as an operational environment.

### 5.6 Summary

The implementation of the Software Development Workbench Executive (SDWE) is realized in three phases. First, the selection of the implementation language(s) used are presented and justified. They are the DEC Command language and PASCAL. Then, the initial implementation of the SDWE Version 1.0 is described as is the update to Version 1.1. This version update is necessary for the SDWE because of the target machine operating system upgrade to VAX-11/780 VMS 3.0.



The extensive theoretical analysis, requirements definition, and design specification for the Software Development Workbench allows the implementation of the SDW to progress with little difficulty. The implementation of the SDWE is especially interesting because of the use of a Command Language, a definite top-down implementation and testing strategy, and the use of the SDW to develop itself.

CHAPTER 6: Integration of the  
Software Development Workbench

### 6.1 Introduction

The Integration stage of the Software Life-Cycle is characterized by the merging together of the systems components into a single system. During this stage, the actual integration of the system components is validated by the testing of the interfaces between the system components.

Within the realm of the Software Development Workbench (SDW) effort, the integration stage involves the joining of the Software Development Workbench Executive (SDWE) to the other SDW components. All of the SDW components are loaded on to a single RK07 disk drive on the VAX-11/780 host computer.

The purpose of this chapter is to define the manner in which the SDW components are installed on the SDW and integrated under the SDW Executive.

### 6.2 Installation of the SDW Components

Prior to the integration of all of the SDW components, each of these components is loaded onto the host computer. The SDWE resides on the host computer, since this computer was used for its development. All of the other SDW components, that are not part of the VMS environment, are loaded onto the VAX-11/780 from magnetic tape or floppy

disk. Since the host computer configuration does not included a tape drive, it is necessary to use some other compatible system in order to perform the transfers from tape to RK07 disk. A PDP-11/34 belonging to Aeronautical Systems Division, Air Force Systems Command, Wright-Patterson AFB, is used for the transfers. Some special procedures are used to facilitate the transfer on the PDP-11/34 machine. Namely, the disk is initialized under the Structure 1 format instead of the regular Structure 2 and the initial directories use strictly numeric names.

Each of the SDW components is transfered to disk using an accompanying installation guide. The Extended Requirements Engineering and Validation System (EREVS) is not included in the initial set of disk resident tools because its transfer uses the VMS "BACKUP" command that is not available on the PDP-11/34. Furthermore, only the executable images and other necessary files are retained on disk for each of the SDW components. A shortage of disk space is responsible for this action.

### 6.3 Integration of the SDW Components and the SDWE

The careful design and implementation of the SDWE allows the efficient integration of the SDWE to the other SDW components. All of the non-VMS SDW components have a

specific logical name defined for their resident disk. The assignment of these logical names is done in the SETSDW.COM procedure that is used to set up for the use of the SDW. Each of the non-VMS SDW components that require special set up activities use a special command module that is callable from the two-letter SDW code for the particular component. Each of these special command modules uses an internal flag, called "INSTALLED", to conditionally execute one of two sections of code. If "INSTALLED" is false, then a message to that effect is displayed on the user's CRT. However, if the flag is true, then the component is set up for and executed. Each of these special command modules is identified by the convention "SET<componentname>.COM". If the component does not require special set up activities, the two-letter code for that component is simply redefined in the "ASSIGNSYM.COM" module to allow for the execution of the component.

Each of the non-VMS SDW components resides in its own directory. The protection on these directories is set to allow their use and each directory is owned by the [200,75] User Identification Code.

The next step in this stage is to test the interfaces between the SDW components. One problem encountered during this process is that the Interim AUTOIDEF and the ICAM Decision Support System reset the default directory in their

internal command procedures. Thus, those command procedures are modified to save and reset the default directory prior to termination of their execution. Besides this minor modification, the integration processes has few problems.

#### 6.4 Installation of the SDW on the Central ICAM Development System

The initial implementation of the SDW is on the AFIT/DEL VAX-11/780, however, the SDW is also installed on the thesis sponsor's Central ICAM Development System (CIDS). This system is also a VAX-11/780. The installation of the SDW on the CIDS simply requires the transfer of the SDWE to the CIDS, since the other SDW components are already resident on that system. The transfer is completed using floppy disks. Since the CIDS runs under Version 2.7 of the VMS operating system, a few minor changes to the SDWE are necessary. These changes include are exclusively in the help facility because Version 2.7 uses a slightly different help facility and the SDW help facility calls it. It is also necessary to set the NPDRS flag to true in order to disallow the use of Project Data Bases. This is required due to the protections placed on the SDWE's resident directory that prohibited the creation of sub-directories by users.

### 6.5 Summary

The integration of the SDW components onto the target computer was a relatively minor operation for two reasons. First, the SDWE was designed to easily accept the incorporation of new tools into the SDW. Second, all of the SDW components were available in VAX versions.

With this final integration and validation of the SDW and its components, the Software Development Workbench is ready to go operation. The following chapter on Operations and Maintenance describes the initial operation phase of the SDW and any maintenance required as a result of the operation experience.

CHAPTER 7: Operations and Maintenance of  
the Software Development Workbench



## 7.1 Introduction

The operations and maintenance stage is the final stage of the software life-cycle. Of primary importance during this stage are the system documentation from the previous phases, the resolution of any problems, and any system modification from new requirements for the system.

For the Software Development Workbench (SDW), this final stage is supported by the assembling of a complete set of documentation for the SDW, the Software Development Workbench Executive (SDWE), and the other SDW components. During this stage, the SDW is first used, as a prototype environment by the AFIT software community. In particular, the system is to be used by the EE7.93 Advanced Software Engineering class and a number of AFIT thesis students. This stage also involves the evaluation of the operational SDW against the criteria established in Chapter 2.

## 7.2 Development of the SDW Documentation Package

Prior to the release of the SDW to the AFIT software community, a complete set of user documentation must be assembled into a SDW Documentation Package. This package includes the SDWE User Manual, the SDWE Maintenance Guide, and the SDWE Installation Guide, as well as the user manuals and installation guides for the other SDW components. The

SDWE documents are included as Appendices I, J, and K.

### 7.3 Maintenance Activities on the SDW

As with most software developments, additional requirements are identified for the system following initial delivery. The SDW is no exception. Four modifications are required to the SDW as a result of user experiences with the system.

The first modification is to the SDW Help Facility. As first implemented, this facility uses a series of queries to provide the user with help on any SDW or VMS command. However, as the user develops greater familiarity with the SDW, he prefers to simply state the help option followed by the specific command code for which he requires help and then receive the appropriate display on his CRT. The SDW Help Facility is modified to provide this capability, as well as the initial walk-through help capability.

The second modification is also a result of additional requirements desired by the user who possess a greater familiarity with the SDW. As is often the case with top-down, menu-driven systems, the user eventually requires the capability to enter a command string at the top level that allows him to directly access tools several levels down the hierarchy (Ref 58). This type of capability requires

the parsing of command strings as entered from the top-level module. The DCL capability to use pre-defined parameters allows the relatively simple modification of the SDW to provide for such command strings.

The third modification to the SDW deals with the usage of Project Data Bases. Previously, the user would not be told whether the Project Data Base specified for use is a new or existing one. As the number of Project Data Bases increases, so does the probability of duplicate names for these data bases. To help avoid these types of problems, the SDWE is modified to report to the user whether the specified Project Data Base is a new or existing one.

The final modification to the SDW deals with the initial and default user's device specifications. The original initial device specification was set to a "VT100". This setting provided for the passing of specific control characters to clear the display and use reverse-video for continue prompts. However, on non-VT100 devices, the specific control characters appeared on the CRT. This looks quite un-professional. Thus, the initial setting of the device specification is set to a "VT52" mode. This mode uses no special control characters and is thus appropriate for any device. The use of the continue prompt between displays is also removed when in the VT52 mode. Since this mode uses scrolling instead of display clearing, the normal

terminal facilities for stopping and continuing scrolling may be used to view the SDW displays.

#### 7.4 Evaluation of the Software Development Workbench

The Software Development Workbench (SDW) only addresses a portion of the specifications for a software development environment presented in chapter 2. While each of the main categories of specifications are addressed, none of these categories are fully satisfied. As a result, it is important to emphasize the strengths and weakness of the SDW to provide a background for future investigations involving the SDW. Although a through evaluation and development of the SDW is not possible, due to the limited time of the investigation, a summary of user reactions to the SDW as an environment is provided. The criteria for this evaluation is established in the requirements definition chapter of the SDW (section 2.5).

7.4.1 Strengths of the SDW. The SDW is found to be a very user-friendly environment. The menu-driven format and extensive help facilities allow the novice user easy access to any SDW component. The SDW is also a very flexible environment because of its capabilities for disabling the auto-menu facility, changing the user's device specification, and providing for the use of command strings.

The SDW implements fail-soft error-handling capabilities. With the currently incorporated tools, the SDW is an excellent aid in the production of software and related documentation.

7.4.2 Weaknesses of the SDW. The SDW possess all of the minimal requirements for a software development environment, as well as a number of extra capabilities. However, these provided capabilities are almost exclusively supportive only of the pre-implementation stages of software development. The SDW possess few if any components to aid and augment post-implementation activities such as code testing and code optimization.

The issue of integration within the environment is only addressed at the higher levels by the SDW. Components are integrated only by means of a common user interface and common data storage locations. There are no means for integrating development data by preserving the relationships between software products and allowing the individual components to share the development data.

### 7.5 Summary

The operational phase of the Software Development Workbench met with a very welcome response. The SDWE proved to be an effective and easy to learn interface to the SDW. The SDW was found to be very helpful for the development of software, especially in the preparation of software documentation and associated models.

Furthermore, the modifications recommended and implemented during this stage greatly added to the power and appearance of the SDW as a product.

CHAPTER 8: Conclusion/Recommendations

### 8.1 Introduction

The purpose of this thesis investigation is two-fold. The initial emphasis of the effort deals with the requirements specification and the development of a design for an interactive and automated software development environment to support the software life-cycle in accordance with accepted software engineering principles. The second emphasis of this investigation is to implement and test a prototype version of a software development environment, the Software Development Workbench (SDW). This prototype serves two purposes. First, it demonstrates the feasibility of some of the requirements and design specifications established during the initial emphasis of the investigation. Secondly, the prototype is actually installed and operational on two distinct development computers to aid in the development of software at these locations.

### 8.2 Design Summary

The Software Development Workbench (SDW) was developed in accordance with a classical version of the software life-cycle (Ref 40:1-5). The first three stages of this life-cycle are requirements definition, preliminary design, and detailed design. Of these three stages, the first two involve the theoretical development of an ideal software



development environment for the Air Force Institute of Technology (AFIT). The third stage, detailed design, is characterized by the detailed specification of a design for the software development environment prototype, the SDW.

The requirements definition stage of this effort involves the definition of the current software development process, a specification of how the process of software development should be addressed, and a study and summary of the concerns and objectives of a state-of-the-art software development environment. The second step in the theoretical development of this investigation is preliminary design. The objective here is to define the structure, required components, and configuration for the software development environment to meet the previously stated requirements.

The detailed design stage bridges the gap between theoretical development and application by specifying the precise design specification for the initial implementation of the SDW. This stage requires the complete development of a top-level user interface and controller for the environment and the careful selection of component tools for the environment. The purpose behind this stage of design is to develop means to realize the theoretical objectives of the previous stages.

### 8.3 Implementation/Test Summary

The second section of this investigation deals with realization of a prototype version of the software development environment specified in the detailed design stage. This prototype is named the Software Development Workbench (SDW). The implementation of the SDW involves the coding and testing of a top-level user interface and controller, called the Software Development Workbench Executive (SDWE), as well as the incorporation of several component tools into the SDW. Following the implementation of the SDWE and the loading of the component tools, all of these SDW sub-systems are integrated to the VAX-11/780 VMS Operating System and to each other. Then, the SDW is completely tested, operating manuals are composed, and the SDW goes into an operational status. The SDW is installed on two distinct host computer systems. The first host is the AFIT/DEL VAX-11/780. Here the SDW is used as a software development aid to support both student course work and Master and Doctorate level research efforts. The second host is the Central ICAM Development System (CIDS). The CIDS facility is used by a number of research groups across the nation (Ref 52). The members of these research groups are themselves developing a very sophisticated development environment for the CIDS that is to become the Integrated Systems Development System (ISDS) upon completion. The SDW is used as a prototype for this much larger environment and

as a tool with which to develop the larger ISDS environment.

#### 8.4 Recommendations for Future Investigations

As one might expect from the previous section and the fact that the SDW is a prototype environment, there are many topics for future investigation dealing with the SDW. A listing of these topics is provided in the following paragraphs. This listing of topics is sequenced in the order in which the topics should most likely be addressed.

8.4.1 Implementation of the Pre-Fab Software Description Data Base. The Pre-Fab Software Description Data Base is a concept for a facility that will greatly reduce the production of software programs and modules that have previously been designed. This capability could represent a very substantial economic and manpower savings. The Pre-Fab Software Description Data Base concept requires a complete development consisting of design and implementation of the facility. Implementation should include the population of the facility with a wide variety of existing software modules.

8.4.2 Enhancement of the SDW Component Set. The SDW currently incorporates only a limited tool set. These tools provide capabilities for a minimal environment with extended capabilities for supporting pre-implementation type

activities. The SDW component set must be enhanced with additional components to support the other functional specifications defined in Appendix C.

8.4.3 Re-Hosting of the Software Development Workbench onto UNIX(TM). The Air Force Institute of Technology has recently procured and installed a larger VAX-11/780 configuration as its new Scientific Support Computer. This computer runs under the UNIX(TM) operating system. The SDWE and the SDW component tools could easily be re-hosted on to this machine by translating all of the DCL command procedures into compatible UNIX command procedures.

8.4.4 Development of an Integrated Project Data Base Schema. At present the Project Data Bases are simply isolated data storage areas. Conceptually, the Project Data Bases should be integrated data storage areas that are managed by a Data Base Management System (DBMS) and preserve the relationships between the development data from the different stages of the software life-cycle. This concept is addressed in greater detail in Section 4.4.

8.4.5 Extension of the SDW's Scope. At present, the SDW provides few facilities to control and coordinate many-programmer projects. The SDW is a software developer's environment and does not provide the managerial capabilities required to manage large software development projects that involve many analysts and programmers. The Air Force is in

great need of such a capability for the large software developments under its direction. In order to provide these needed capabilities, the SDW must utilize sophisticated configuration management and planning tools. These tools must be able to interface to and retrieve development data from the other SDW components.

8.4.6 Development of a Consistency- and Syntax-Directed Editor. A syntax-directed editor uses a Backus Normal Form definition of a language's syntax to automatically check the syntactical correctness of programs as they are entered on to the computer (Ref 33). This capability could be extended so that the editor also accepts a previous specification for the software system, such as a requirements or design specification. Thus, the editor checks for both syntactical correctness and consistency with the previously stated specifications for the software system.

## Bibliography

1. Ada Reference Manual, Department of Defense.
2. Alford, M. W. "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering (Jan 1977).
3. Alford, M. W., Irby, J. E., Scott, J. E., Lawson, J. T., Osborne, R. G., Hardy, E. J. Distributed Computing Design System. Huntsville, AL : TRW Defense and Space Systems Group, August 1981.
4. Andrews, D. M. and Melton, R. A. FORTAN Automated Verification System (FAVS) User's Manual. Rome Air Development Center, RADC-TR-78-268, January 1979.
5. Appleton, Daniel S. "Measure Twice, Cut Once," DATAMATION (February 1982).
6. Atwood, M. E., The Processes Involved in Designing Software. AD-A092935/6, August 1980.
7. Baker, F. T. "Structured Programming in a Production Programming Environment," Proceedings, 2nd International Conference on Reliable Software (April 1975).
8. Balzer, R., Goldman, N. and Wile, D. "On the Transformational Implementation Approach to Programming," Proceeding, 2nd International Conference on Software Engineering, IEEE: Long Beach, Calif. 1976.
9. Basili, Victor R. "FLEX: A Flexible, Automated Design System," AD A079312.
10. Bergland, Glenn D. and Gordon, Ronald D. Tutorial: Software Design Strategies. 2nd Ed. Murray Hill, NJ : Bell Laboratories, 1981.
11. Bianchi, M. H. and Wood, J.L. "A User's view on the Programmer's Workbench," Proceeding, 2nd International Conference on Software Engineering, Oct 1976.
12. BMDATC Software Development System, Vol 1. Ballistic Missile Defense Advanced Technology Center, AD-B014-623.
13. Boehm, Barry W. "Module Design and Interface Validation," IEEE Transactions on Software Engineering, Jan 1975.

14. Boehm, Barry W. "Some Experience With Automated Aids to the Design of Large Scale Reliable Software," IEEE Transactions on Software Engineering, Vol. 1, No. 1 (March 1975).
15. Branstad, M. A. and Adrion W. R. "NBS Programming Environment Workshop Report," Special Publication 550-78, Institute for Computer Science and Technology, National Bureau of Standards, Washington D.C., 1981.
16. Bratman, H. and Court, T. "The Software Factory" COMPUTER 8 (May 1975) p 28-37.
17. Bratman, H. "Automated Techniques for Project Management and Control". Practical Strategies for Developing Large Software Systems, Addison-Wesley, Reading, Mass. 1975.
18. Buxton, J. N. "An Informal Bibliography on Programming Support Environments," SIGPLAN Notices (December 1980) page 17.
19. Chrusciki, Andrew; Simpson, Louis; and Sheffield, R. JOVIAL J73 Programming Support Library, Rome Air Development Center, RADC-TR-82-162, June 1982.
20. Chyuan-Shiun Lin, "A Structured Functional Testing Approach," NCR.
21. Clarke, L. A. "The Source Code Control System". IEEE Transactions on Software Engineering, (Jan 1977).
22. Conrad, Thomas P. "Application of Advanced Software Technology to Submarine Command and Control," Naval Underwater Systems Center, Newport, RI. AD-B050-288L.
23. Cotterman, William W., Couger, J. Daniel, Enger, Norman L., Harold, Frederick, Systems Analysis and Design : A Foundation for the 1980s. New York : North Holland Publ., 1981.
24. Davis, C. G. and Vick, C. R. "The Software Development System". IEEE Transactions on Software Engineering, (Jan 1977).
25. Davis, Richard M. Thesis Projects in Science and Engineering. New York, New York : St. Martin Press, 1980.
26. Denning, Peter, "Parts Based Programming," Computer Science Department, Purdue University, IEEE COMPCON, 1981.

27. Digital Equipment Corporation, VAX/VMS Command Language User's Guide, March 1980.
28. "DoD Requirements for Ada Programming Support Environments, STONEMAN," HOLWG, February 1980.
29. Dolotta, T. A. and Mashey, J. R. "An Introduction to the Programmer's Workbench". Proceedings, 2nd International Conference on Software Engineering, Oct 1976.
30. Duvall, Data and Analysis Center for Software, AD-A089678/7, June 1980.
31. Features of Software Development Tools, National Bureau of Standards. PB81-176562, Feb 1981.
32. Glass, Robert L. "Persistent Software Errors," IEEE Transactions on Software Engineering, Vol. SE-7, No. 2, (March 1981).
33. Gutz, Steve, Wasserman, A. and Spier, M., "Professional Development System for the Professional Programmer," COMPUTER, Vol. 14, No. 4 (April 1981) p45-53.
34. Hamilton, M. and Zeldin, S. "High Order Software -A Methodology For Defining Software". IEEE Transactions on Software Engineering (Jan 1977).
35. Hausen, H. and Mullenburg, M. "Conspectus of Software Engineering Environments," 5th International Conference on Software Engineering, San Diego, CA March 1981, IEEE Catalog No. 81CH1627, pp. 34-43.
36. Houghton, National Bureau of Standards Software Tools Data Base, National Bureau of Standards, PB81-124935, 1981.
37. Howden, William E. "Applicability of Software Validation Techniques to Scientific Programs" ACM Transactions on Programming Languages and Systems. 2,3(1980).
38. Howden, William E. "Contemporary Software Development Environments," Communications of the ACM, Vol 25, No 5 (May 1982).
39. Howden, William E. "Functional Testing and Design Abstraction," Journal of Systems and Software, 1980.
40. Hunke, Horst, Software Engineering Environments.



New York : North Holland Publ., 1981.

41. ICAM/SEM Coalition Program Meeting Notes from the 1-4 March project review at Wright-Patterson AFB.
42. Integrated Computer Aided Manufacturing, Dynamic Modeling Manual (IDEF-2). Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH, June 1981.
43. Integrated Computer Aided Manufacturing, Function Modeling Manual (IDEF-0). Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH, June 1981.
44. Integrated Computer Aided Manufacturing, Information Modeling Manual (IDEF-1). Materials Laboratory, Air Force Wright Aeronautical Laboratories, Air Force Systems Command, Wright-Patterson AFB, OH, June 1981.
45. Ivie, E. L. "The Programmer's Workbench - A Machine for Software Development," CACM, Vol. 20, No. 10 (October 1977) pp. 746-753.
46. Jefferies, Robin "The Process Involved in Designing Software," University of Colorado, AD A092 935.
47. Keringhan B. W. and Mashey, J. R. "The UNIX Programming Environment," COMPUTER, Vol. 14, No. 4, p. 12 (April 1981).
48. Kernighan B. W. and Plauger, P. J. Software Tools. Addison Wesley, Reading MA, 1976.
49. Kernighan B. W. and Plauger, P. J. "Software Tools". Proceedings, 1st National Conference on Software Engineering, Sept 1975.
50. Lamergon, Robert G. and Dugan, Dennis K. "Software Engineering With Reusable Designs and Code" IEEE COMPCON 1981.
51. Loshbough, R. P. Applicability of SREM to the Verification of Management Information System Software Requirements, AD-A100720/2 and AD-A100721/0.
52. Mashey, J. R. and Smith, D. W. "Documentation tools and Techniques". Proceeding, 3rd International Conference on Software Engineering (Oct 1976).

53. Mayer, Richard, "Unified SEM: The ICAM Approach to Systems Software Development," Proceedings COMPSAC 79, Chicago, IL November 1979.
54. Melton, Richard; Greenburg, Gary; and Sharp, Michael. COBOL Automated Verification System: Study Phase, Rome Air Development Center, RADC-TR-81-11, March 1981.
55. Millington, D. Systems Analysis and Design for Computer Applications. New York : Ellis Horwood Limited, 1981.
56. Mullens, Dan E. Investigation of Meta-Language Modelling for Translation between Simulation Languages and Requirement Definition Languages, AFIT/GCS/EE/82M-4.
57. Myers, Glenford J. Software Reliability: Principles and Practice, New York:Wiley-Interscience Publ. 1976.
58. Nusinow, E. I. and O'Connor, Fran, Integrated Systems Development System Needs Analysis Document, NAD170132000. Dayton, Ohio : Control Data Corporation, February 1982.
59. Osterweil, L., A Software Life Cycle Methodology and Tool Support, AD-A076335/9, April 1979
60. Osterweil, L., "Software Environment Research: Directions for the Next Five Years," Computer, Vol. 14 No. 4 (April 1981) p35-43.
61. Osterweil, L., "Using Data Flow Tools in Software Engineering," AD A076 300/6.
62. Radatz, Jane W. "Analysis of IV&V Data," Logicon, Inc. RADC-TR-81-145, June 1981.
63. Ramanoorthy, C. U. and Ho, S. E., "Testing Large Software With Automated Software Evaluation Systems". IEEE Transactions on Software Engineering, January 1977.
64. Reifer, D. J. and Trattner, S. "A Glossary of Software Tools and Techniques". IEEE Transactions on Software Engineering, January 1977.
65. Riddle, William E. "An Assessment of DREAM," N80-30066/8.
66. Riddle, William E. "Flight Software Requirements and Design Support Software," N80-30061.
67. Riddle, William E. "Software Development Environments: Present and Future," N80-30065/0.

68. Ritchie, D. M. and Thompson, K. L. "Special issue dedicated to the UNIX Time-Sharing System", The Bell System Technical Journal, Vol. 57, No. 6, Part 2, (July-August 1978), pp. 1897-2304.
69. Ritchie, D. M. and Thompson, K. L. "The UNIX Time-Sharing System," CACM, Vol. 17, No. 7 (July 1974), pp. 365-375.
70. Robertson, P; Melton, R.; and Andrews, C. COBOL Automated Verification System User's Manual, General Research Corporation: Santa Barbara, CA, May 1982.
71. Rochkind, M. J. "The Source Code Control System". IEEE Transactions on Software Engineering, Dec 1975.
72. Rochkind, M. J. "The Source Code Control System". IEEE Transactions on Software Engineering, January 1977.
73. Ross, D. T. and Schoman, K. E. "Structured Analysis For Requirements Defination". IEEE Transactions on Software Engineering, January 1977.
74. Satterfield, Doyce. "Overview of Software Production Tools," Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama.
75. Schindler, M. "Software Productivity Needs Tools For Improvement," Electronic Design, Vol.28, No.17 (16 August 1980) p45-8.
76. Schneider, Hans-Jochen. and Wasserman, Anthony I. Automated Tools for Information Systems Design, North-Holland Publishing Co.: Amsterdam, Holland, 1982.
77. Smith, Paul. FORTAN CODE AUDITOR User's Manual, TRW: Redondo Beach, CA, Dec 1976.
78. Software Research Associates. Automated Tools for Software Engineering Seminar, Software Research Associates: San Francisco, CA, October 1980.
79. Softech, Inc. "An Introduction to SADT: Structured Analysis and Design Technique," Softech, Inc.:Waltham, Mass, 1976.
80. "Software Automation Attacks the Programmer Bottleneck," Electronic Design (12 Nov 1981) p11.
81. Stephens, S. A. and Tripo L. L. "Requirements Expression and Verification Aid". Proceedings, 3rd International Conference on Software Engineering, May 1978.

82. Sutton, S. A. and Basil, V. R. FLEX: A Flexible, Automated Design System, AD-A079312/5.
83. Teichroew, D. and Hershey, E. A. "PSL/PSA: A Computer-Aided Technique For Structured Documentation and Analysis," IEEE Transactions on Software Engineering, January 1977.
84. Teitelbaum, Tim. The Cornell Program Synthesizer: A Tutorial Introduction, TR 79-381, July 1979, Revised January 1980, Dept. of Computer Science, Cornell University, Ithaca, NY.
85. Teitelbaum, Tim, "The Why and Wherefore of the Cornell Program Synthesizer", Proceedings of the Symposium on Text Manipulation, SIGPLAN and SIGOA, Portland, OR, 1981.
86. TRW, A New Approach For Software Success. Redondo Beach, California : TRW, Inc. 1982.
87. UCSD Pascal Users Manual.
88. Walker, Michael G., Managing Software Reliability. New York : North Holland Publ., 1981.
89. Wasserman, A., "Automated Development Environments," Computer, Vol. 14 No. 4 (April 1981) p7-10.
90. Weinberg, Victor. Structured Analysis. New York, New York : Yourdon Press, 1978.
91. Willis, R. R. "DAS- An Automated System to Support Design Analysis". Proceeding, 3rd International Conference on Software Engineering, May 1978.
92. Wood, R. J. Computer Aided Program Synthesis, University of Maryland, AD-A092621/2, Jan 1980.
93. Yourdon, Edward & Constantine, Larry L., Structured Design, 2nd Ed., New York, New York: Yourdon Press, 1978.

Appendix A: A Model of the Existing  
Software Development Process

## A Model of the Existing Software Development Process

The Software Development Workbench (SDW) is a software development environment that utilizes automated and interactive tools to support the Software Life-Cycle. In order to achieve this goal, a thorough understanding of the existing life-cycle is required. The Structured Analysis and Design Technique (SADT) model of this chapter is a vehicle for gaining this understanding of the life-cycle. The model represents a generic view of the life-cycle as it exists today. In reality the life-cycle is defined and realized many different ways. The variety that exists in these many versions of the life-cycle is attenuated in the model by dealing with the life-cycle stages and component activities in broad terms.

The objectives of the model are to provide a generic view of the life-cycle, to identify areas that require automated support, and to realize where the life-cycle needs to be modified to improve the efficiency of development and the reliability of the product. The model does not consider issues of resource allocation or planning that are characteristic of the larger development efforts.

The actual "As-Is" model is preceded by a diagram listing of all of model's component diagrams. This listing is on the next page and is followed by the actual model.

DIAGRAM LISTING OF THE "AS-IS" MODEL

-----

Number:    Node:    Title:  
-----

SDW01	A-0	Perform Software Life-Cycle (Context)
SDW02	A0	Perform Software Life-Cycle
SDW03	A2	Understand the Problem
SDW04	A21	Conduct Needs Analysis
SDW05	A22	Perform Requirements Definition
SDW06	A223	Define Requirements
SDW07	A225	Review Requirements Document
SDW08	A3	Formulate the Solution
SDW09	A31	Perform Preliminary Design
SDW10	A331	Define Interfaces
SDW11	A312	Develop a Valid Top-Level Design
SDW12	A313	Develop Intermediate-Level Designs
SDW13	A32	Perform Detailed Design
SDW14	A33	Synthesize Implementation & Test Strategy
SDW15	A4	Construct & Integrate Solution
SDW16	A41	Convert Solution Design into an Executable Image
SDW17	A42	Run Test Cases
SDW18	A5	Operate & Maintain Software
SDW19	A52	Operate Software System
SDW20	A54	Correct Software Problem

USED AT:	AUTHOR: HADFIELD PROJECT: SDU	DATE: 11/17/82 REV: 1	WORKING DRAFT RECOMMENDED X PUBLICATION	READER	DATE	CONTEXT:
NOTES: 1 2 3 4 5 6 7 8 9 10						

DEADLINES

PERFORM  
SOFTWARE  
LIFECYCLE

SOFTWARE  
CONCEPT

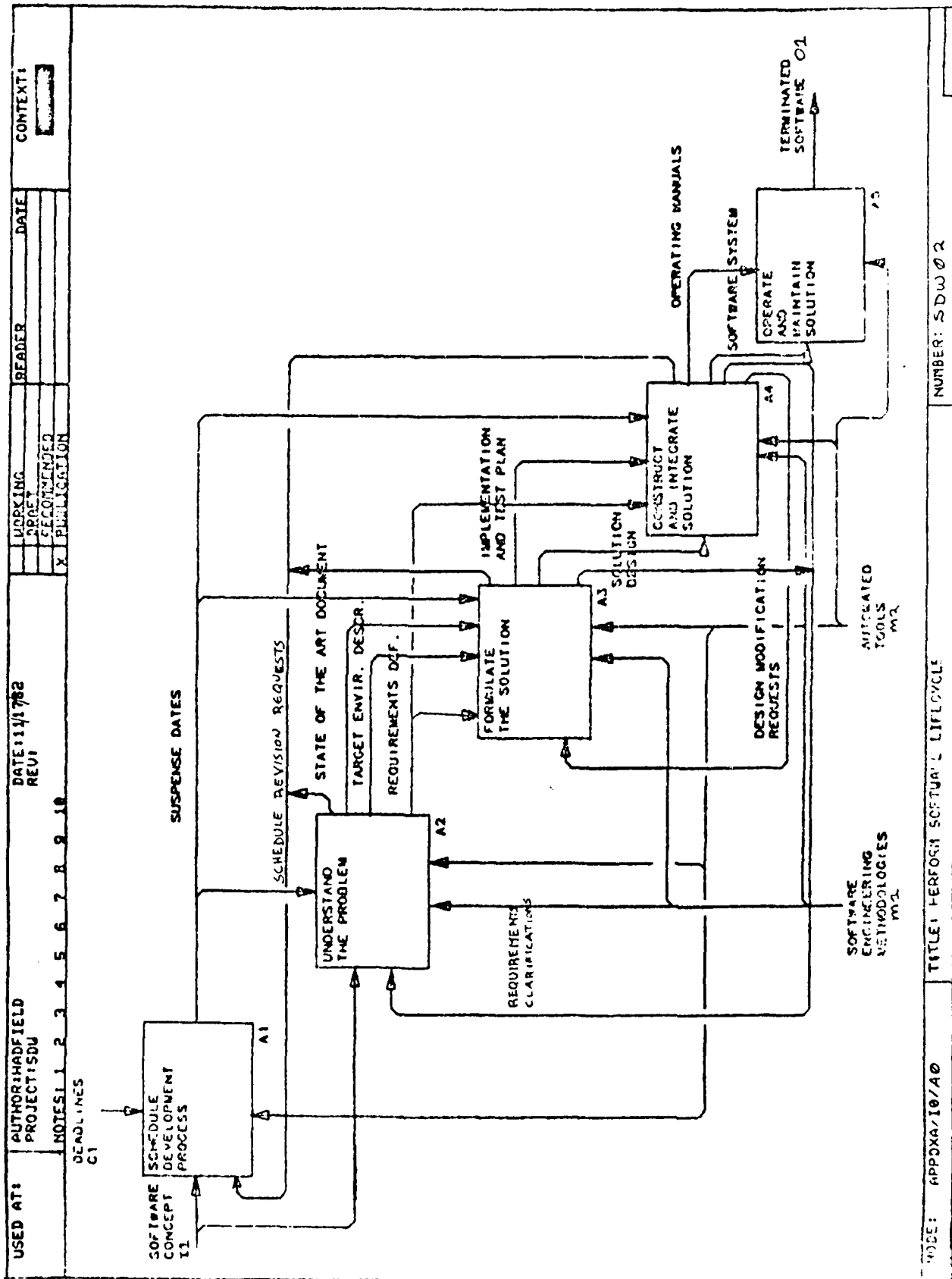
TERMINATED  
SOFTWARE

SOFTWARE  
ENGINEERING  
METHODOLOGIES

AUTOMATED  
TOOLS

MODE: APPDXA/10/A-0	TITLE: PERFORM SOFTWARE LIFE CYCLE	NUMBER: SDW 0 1
---------------------	------------------------------------	-----------------

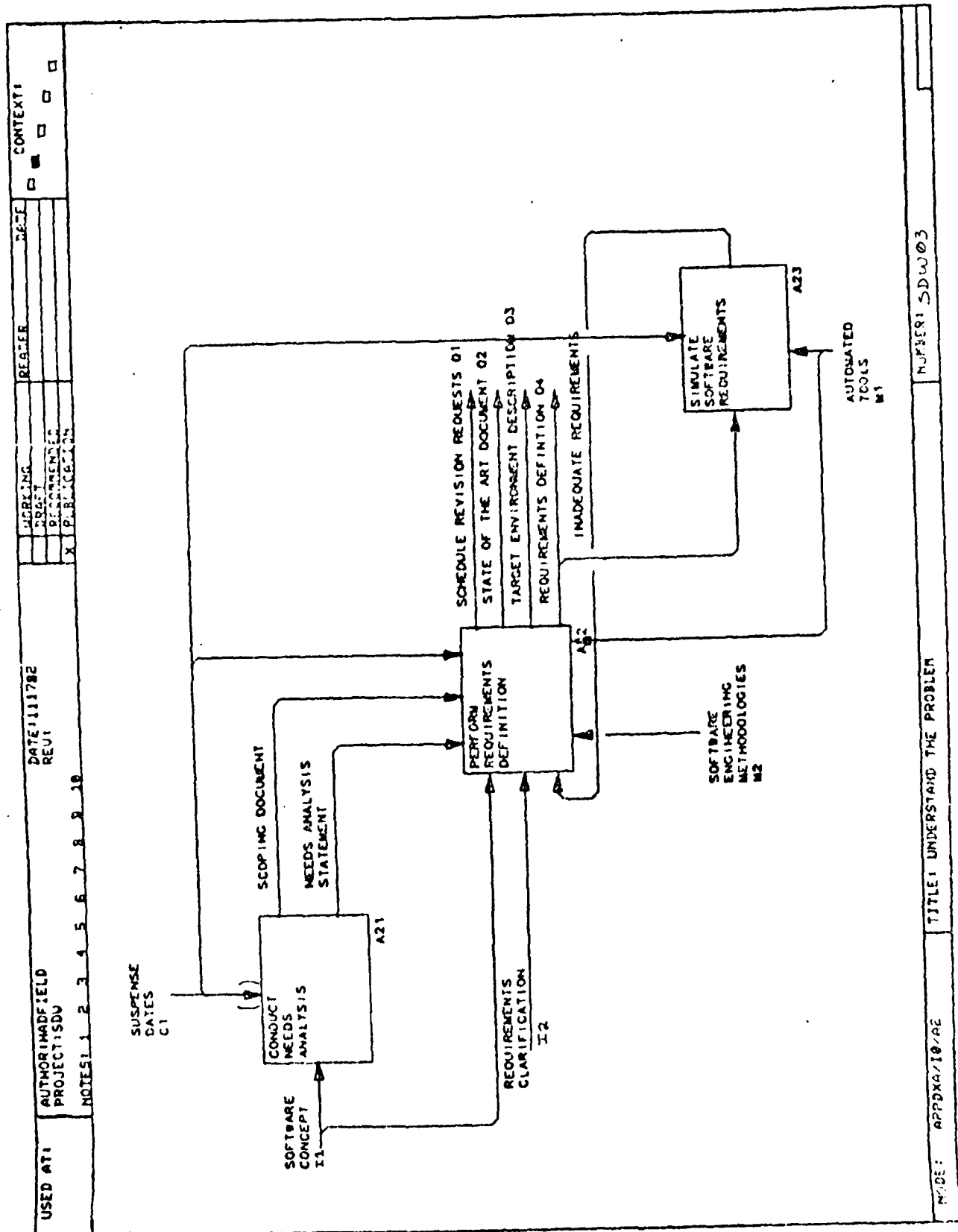




TITLE: PERFORM SOFTWARE LIFECYCLE

MODE: APPDXA/10/A0

NUMBER: SDW02



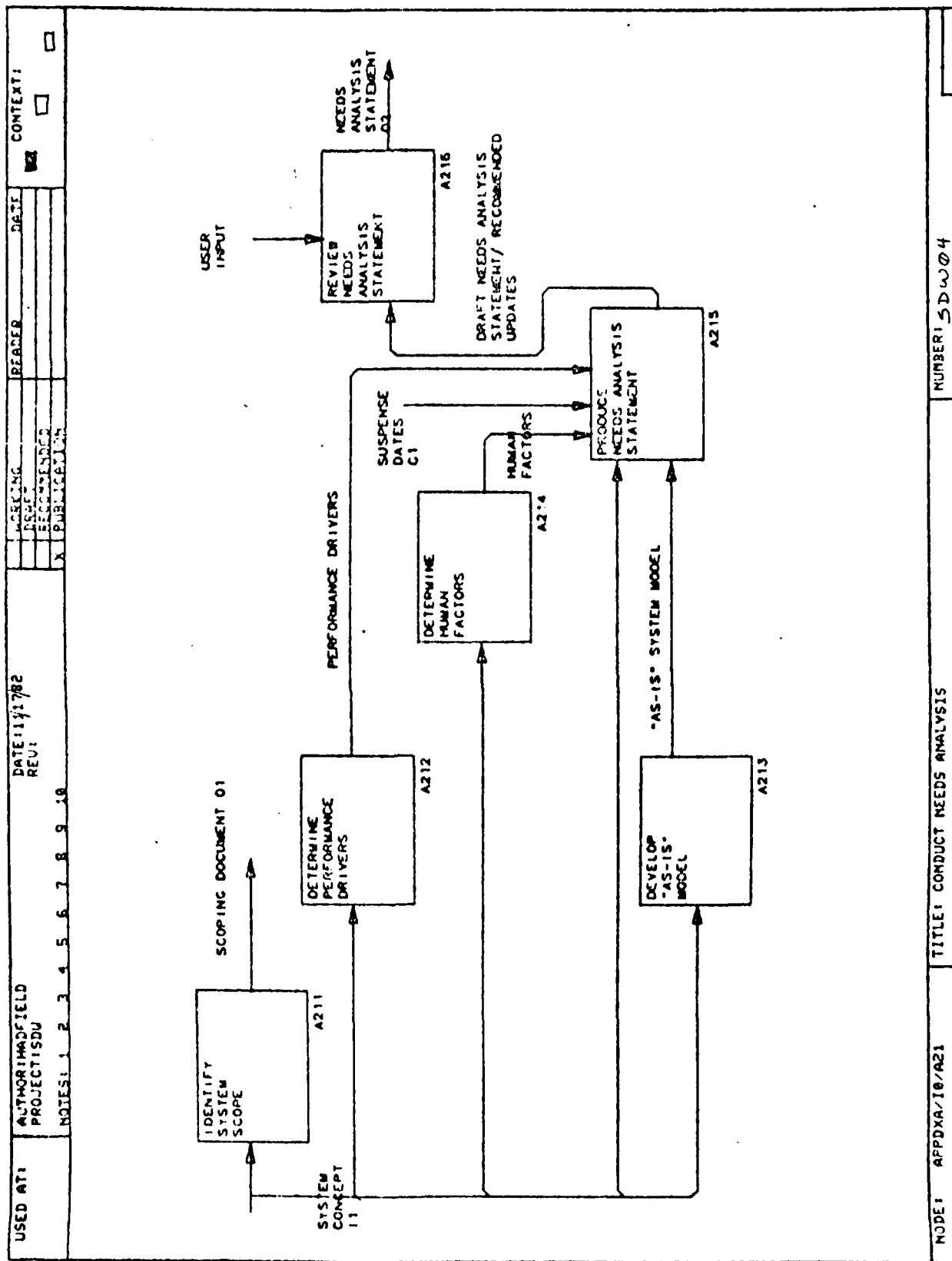
SCHEDULE REVISION REQUESTS 01  
STATE OF THE ART DOCUMENT 02  
TARGET ENVIRONMENT DESCRIPTION 03  
REQUIREMENTS DEFINITION 04

INADEQUATE REQUIREMENTS

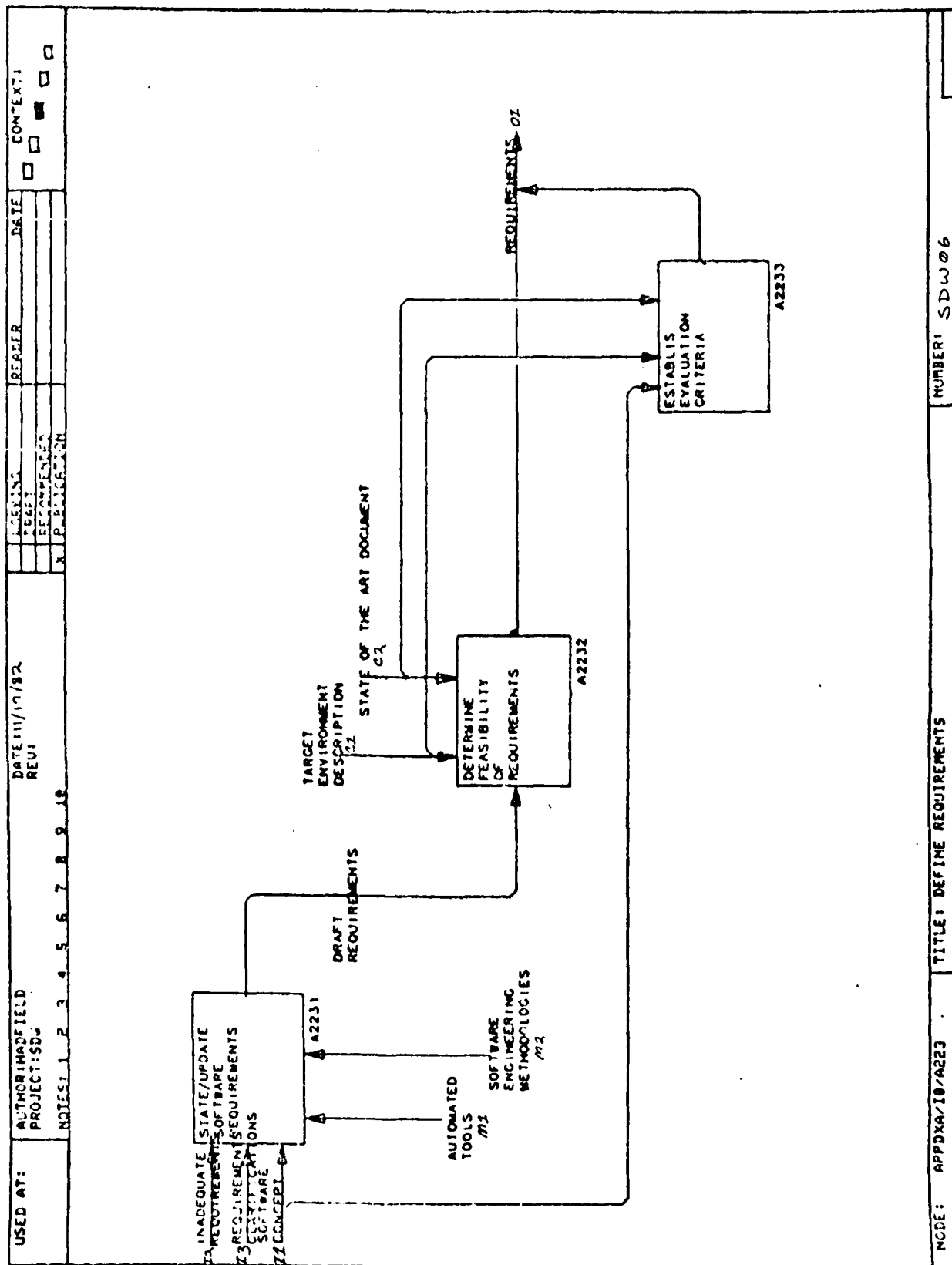
NUMBER: SDW003

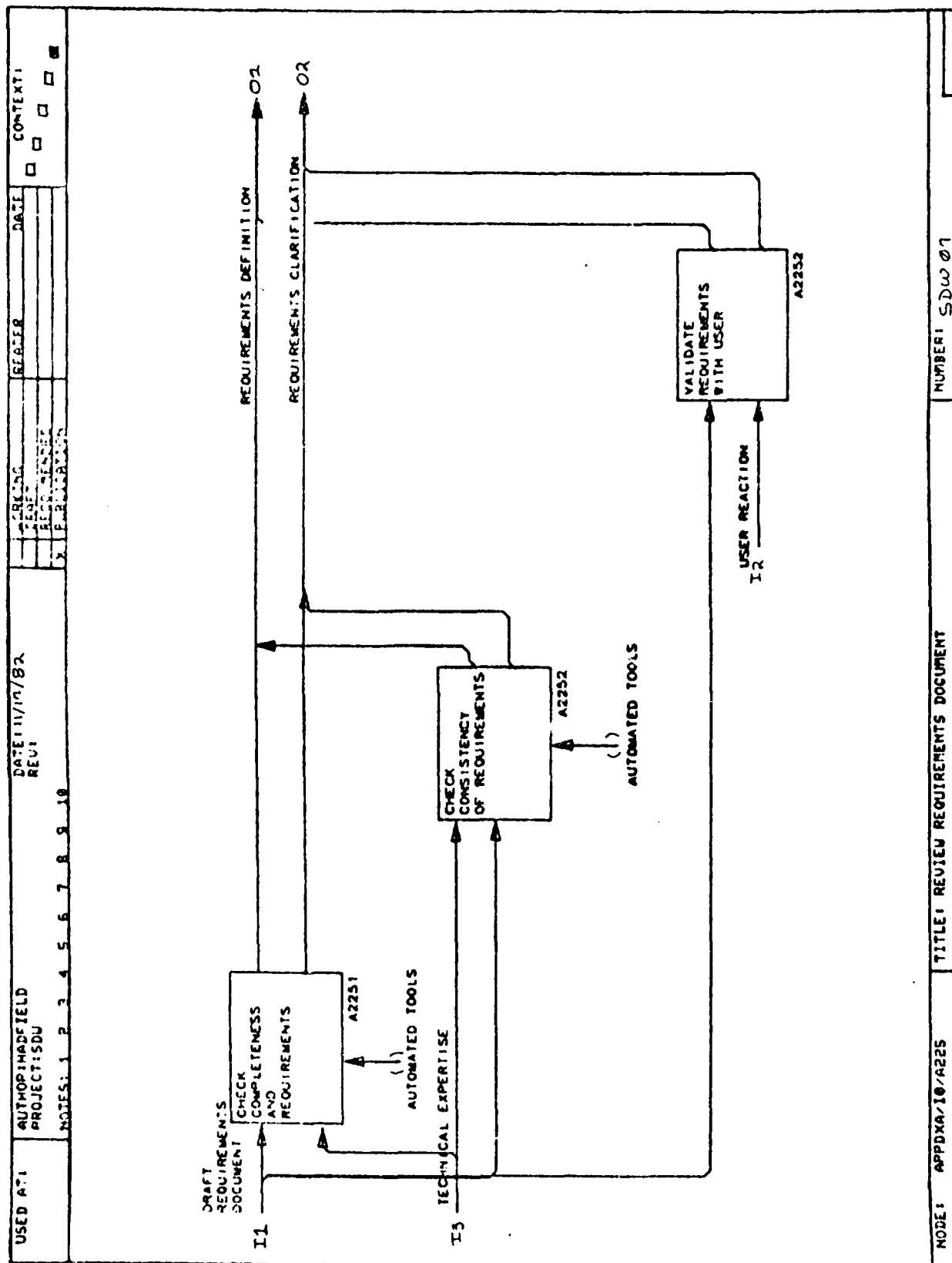
TITLE: UNDERSTAND THE PROBLEM

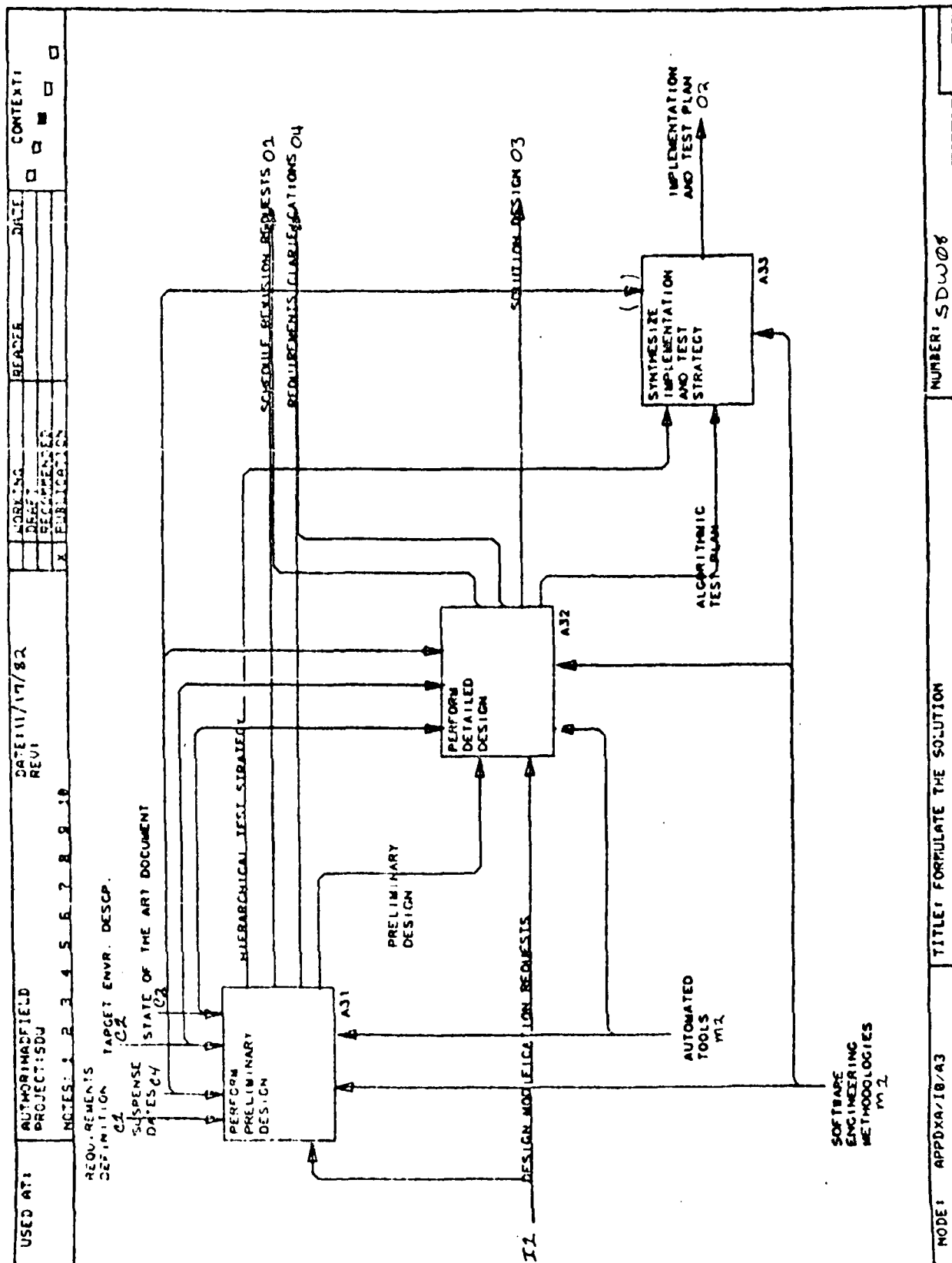
MODE: APPDXA/10/A2



[illegible]



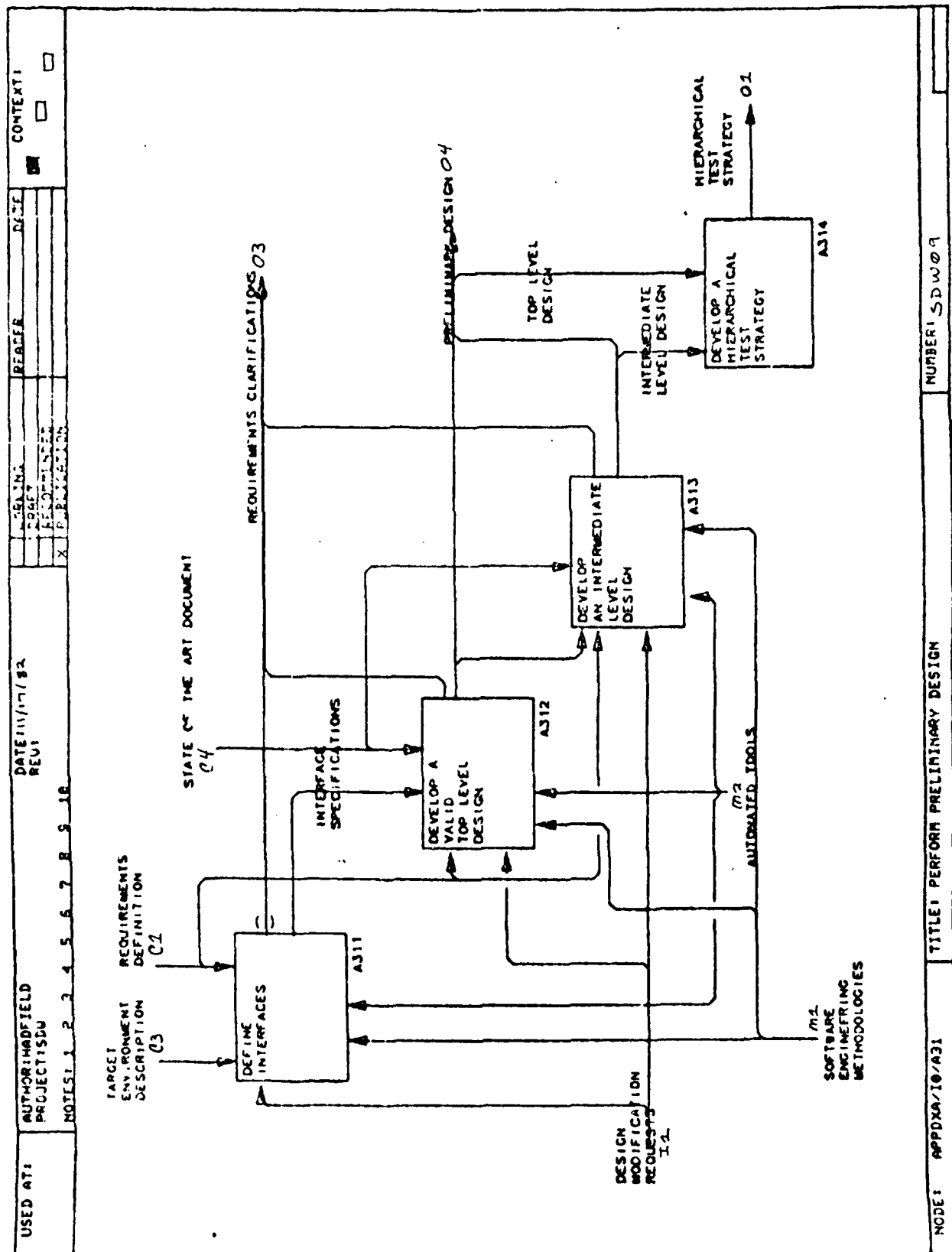




NUMBER: SDW08

TITLE: FORMULATE THE SOLUTION

MODE: APPDXA/18/43

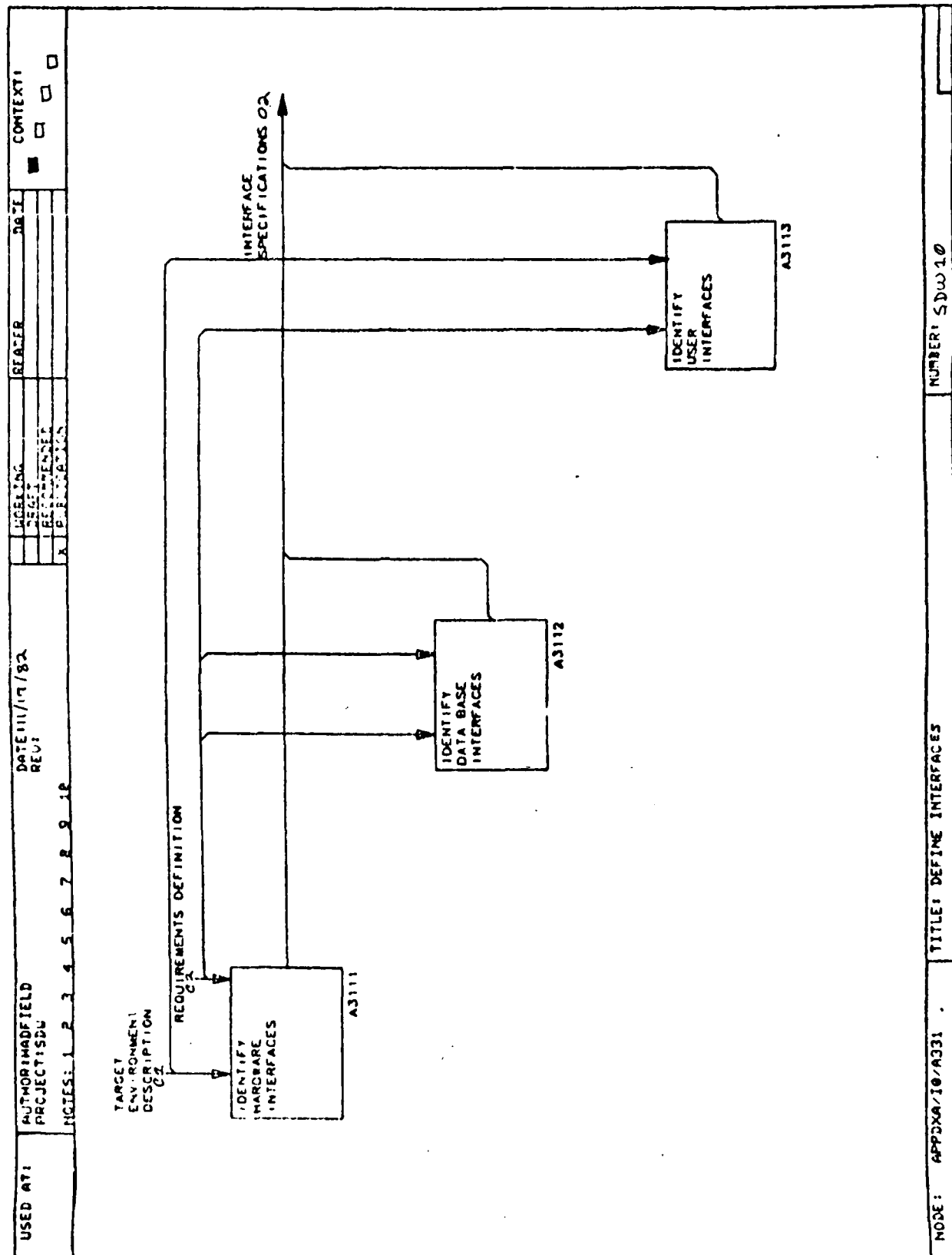


NUMBER: SDW09

TITLE: PERFORM PRELIMINARY DESIGN

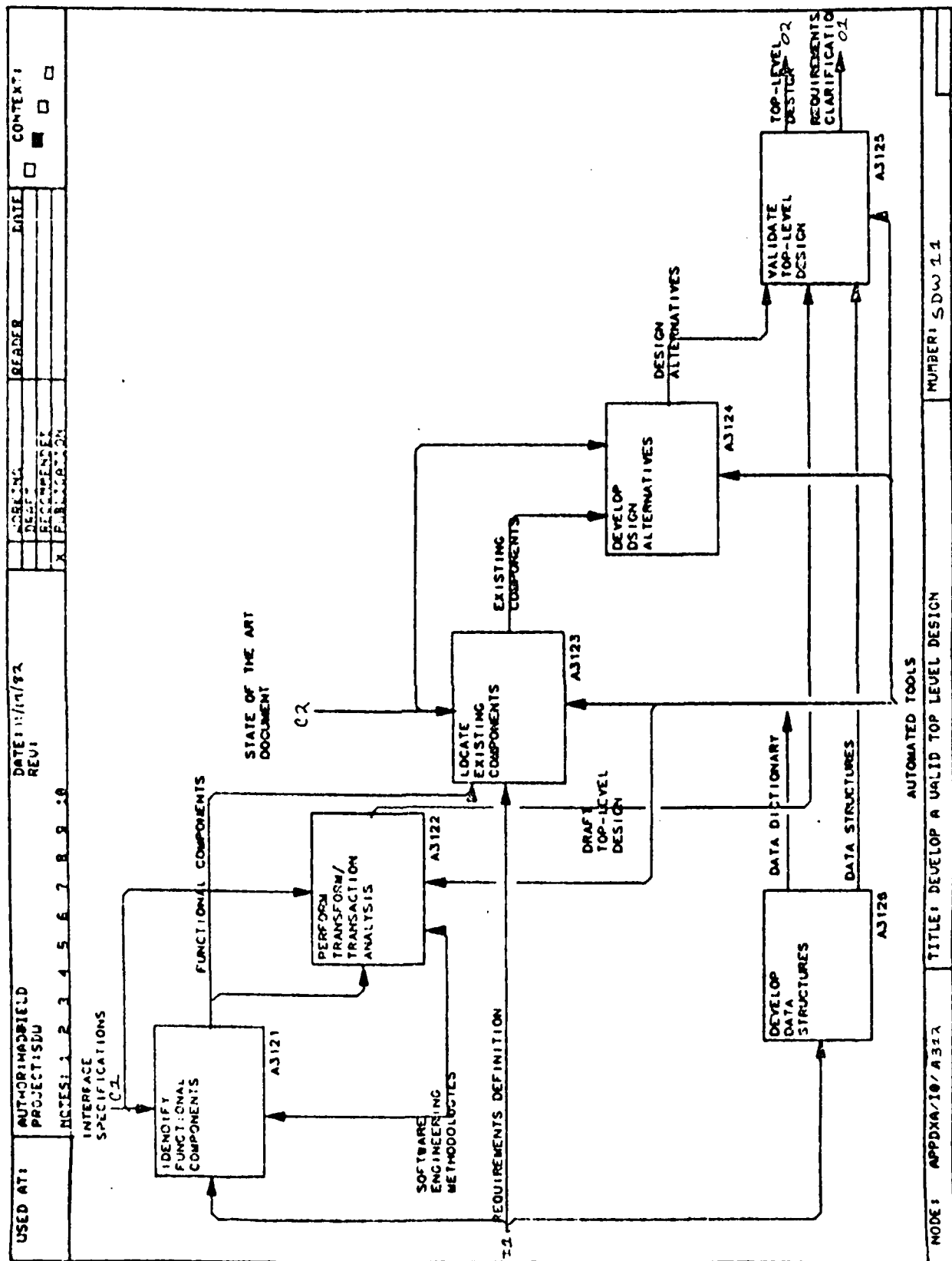
NODE: APPD0A/10/A31

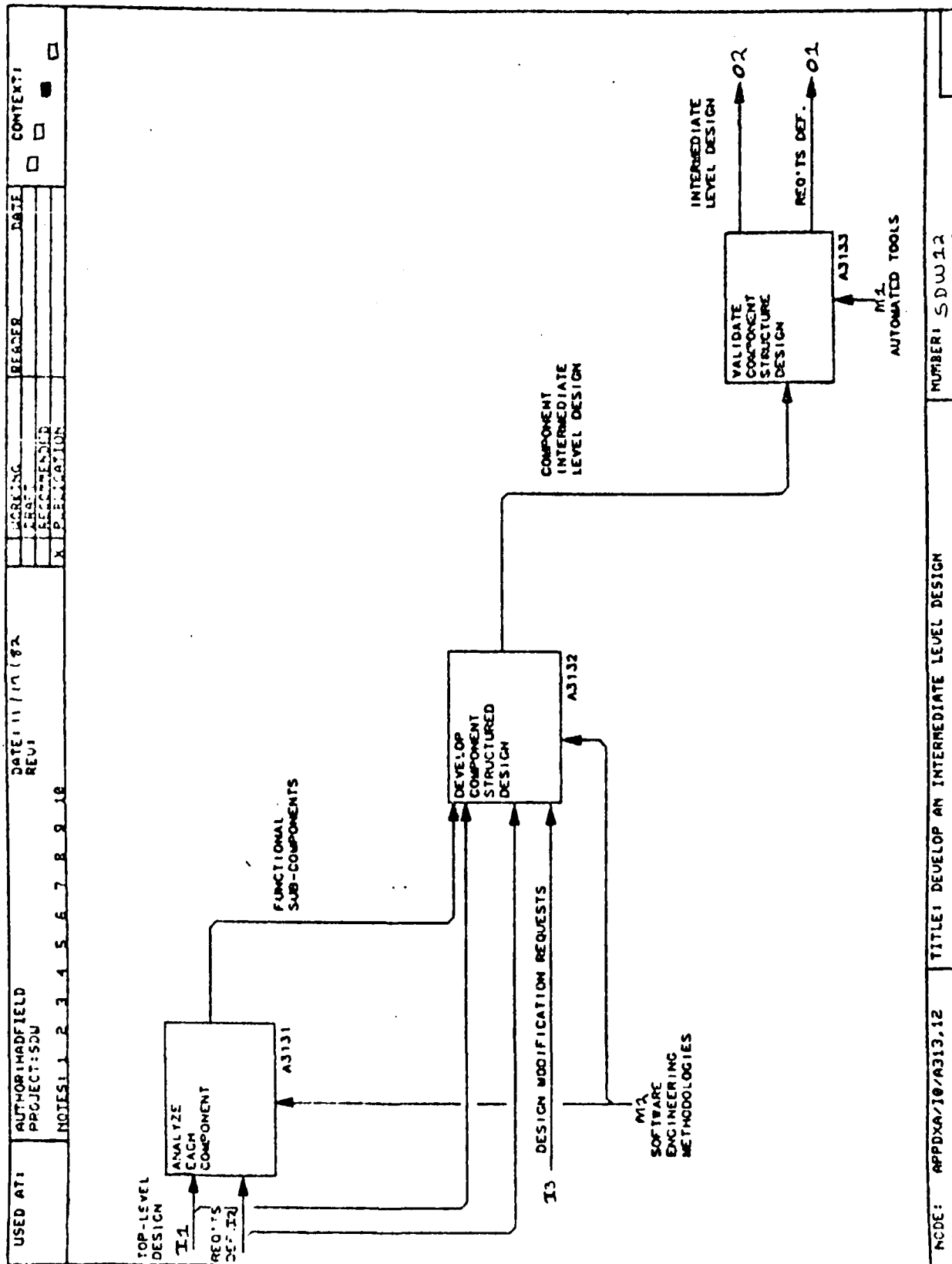




NODE: APPDWA/10/A331 . TITLE: DEFINE INTERFACES

NUMBER: SDW 10

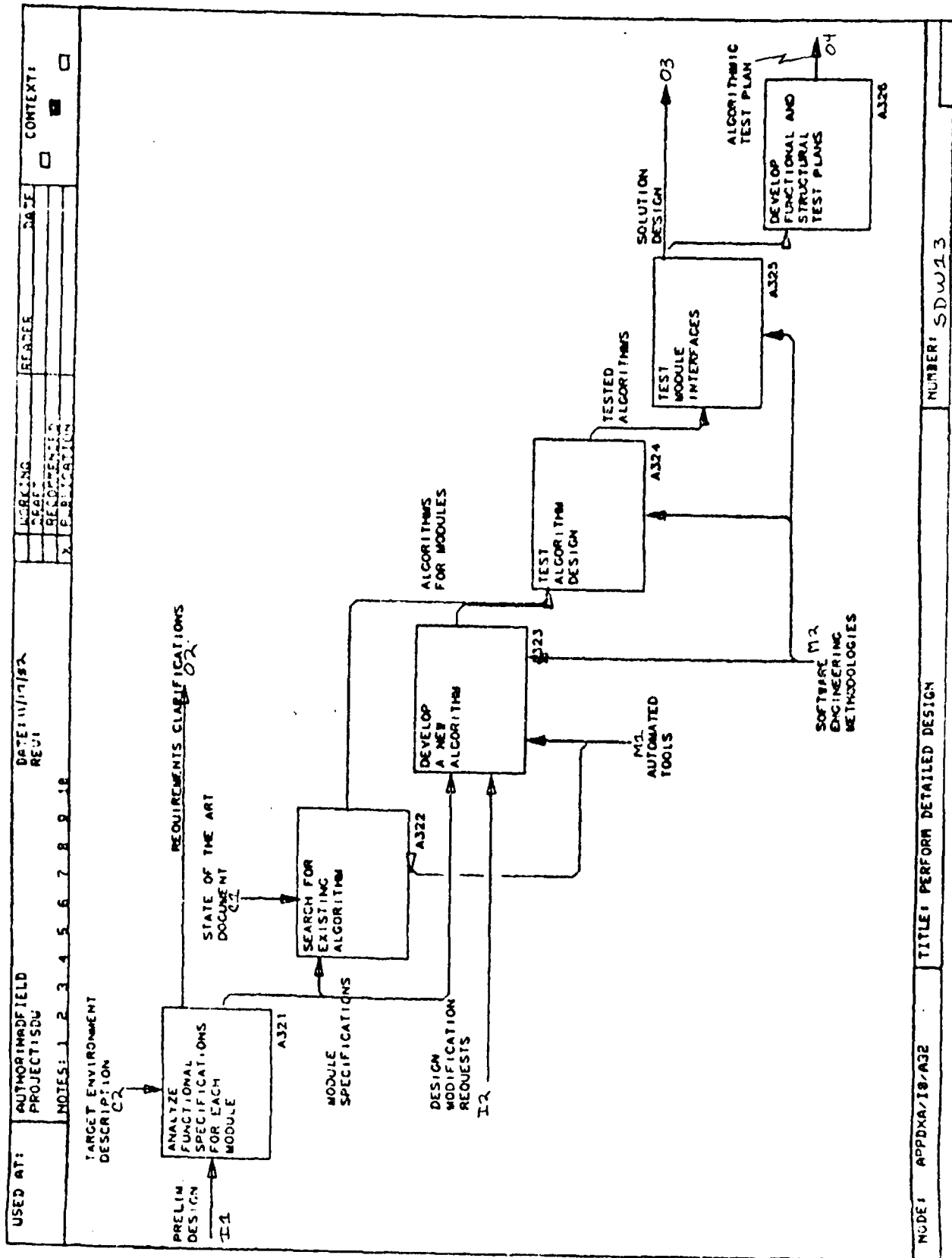


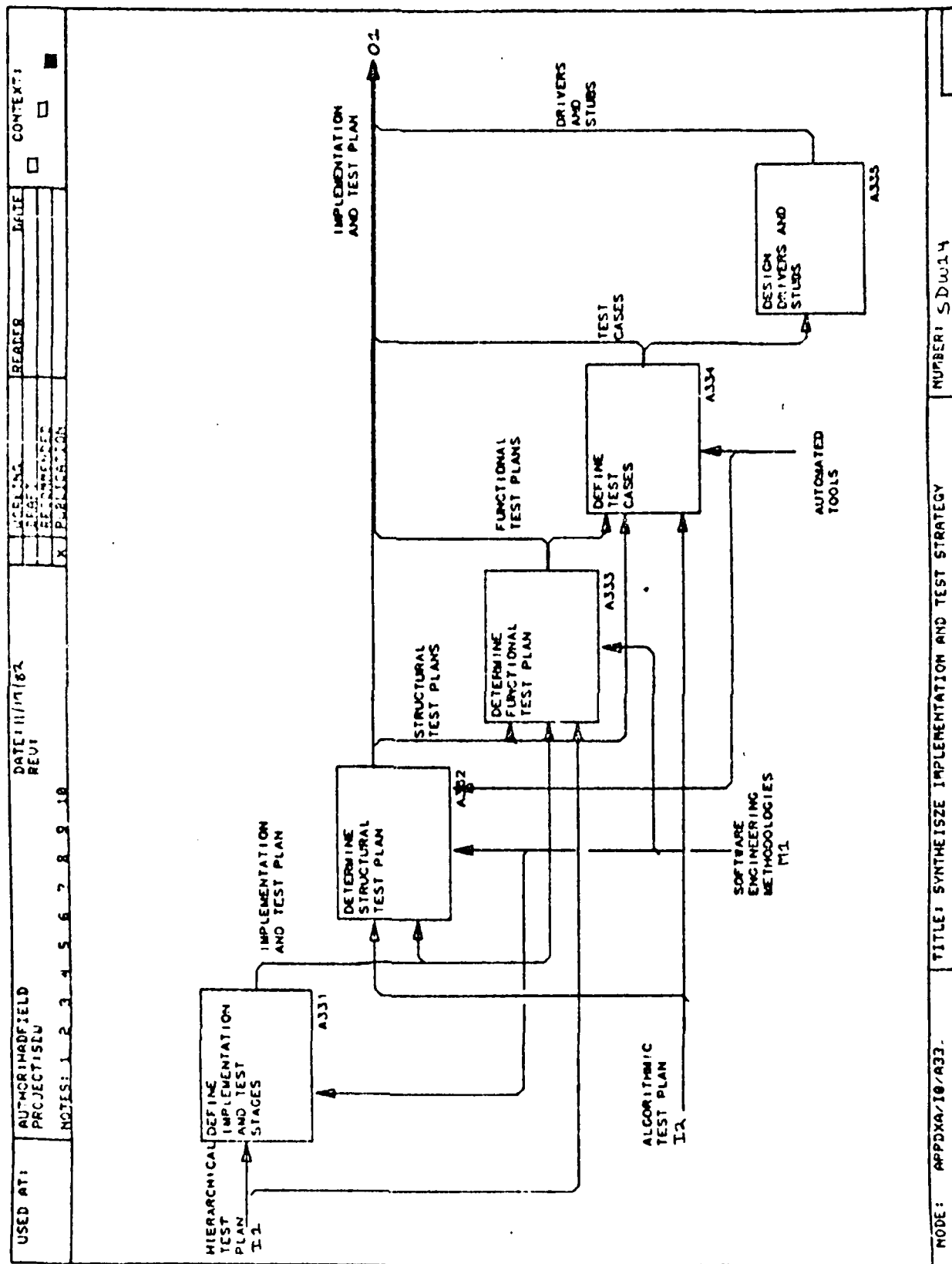


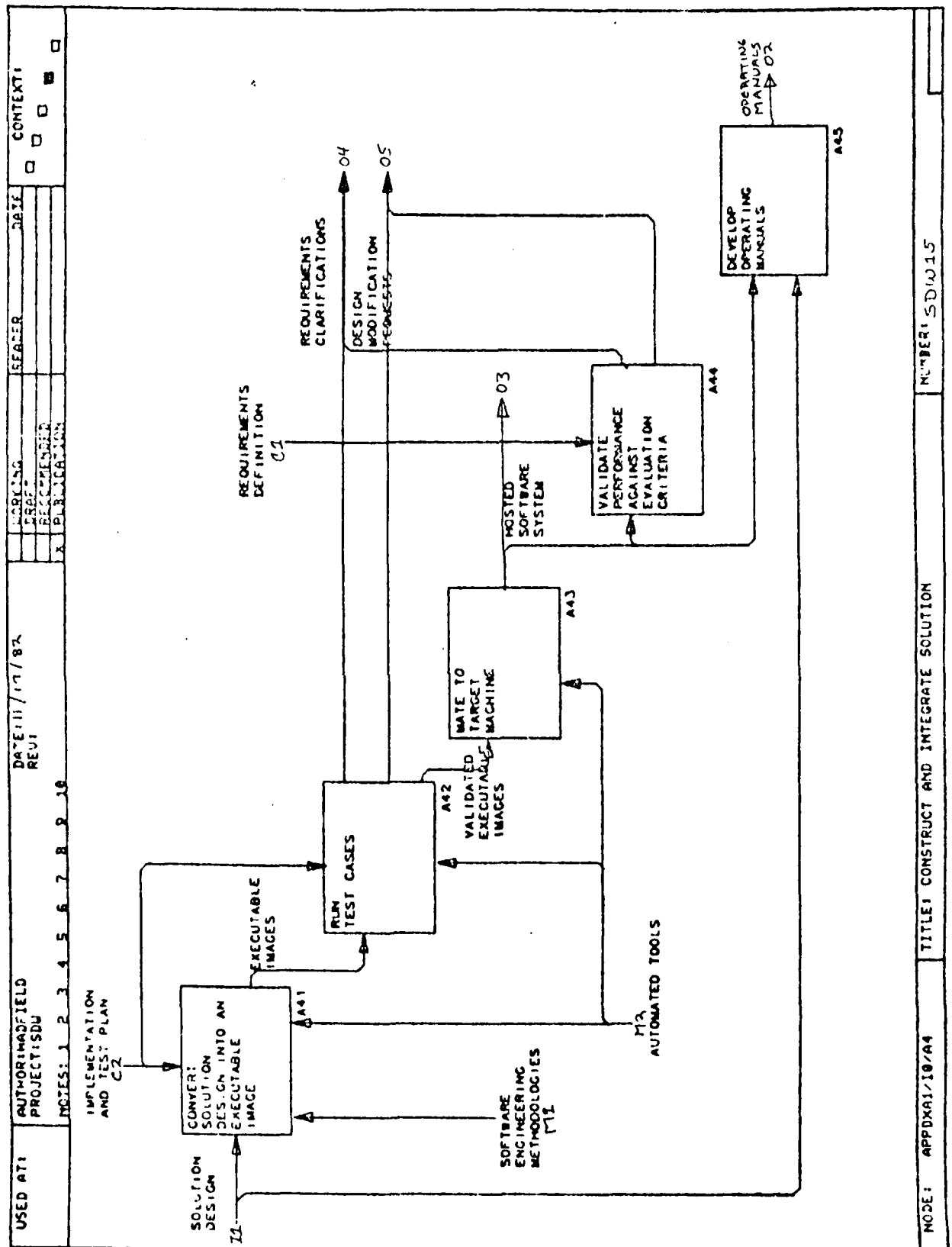
TITLE: DEVELOP AN INTERMEDIATE LEVEL DESIGN

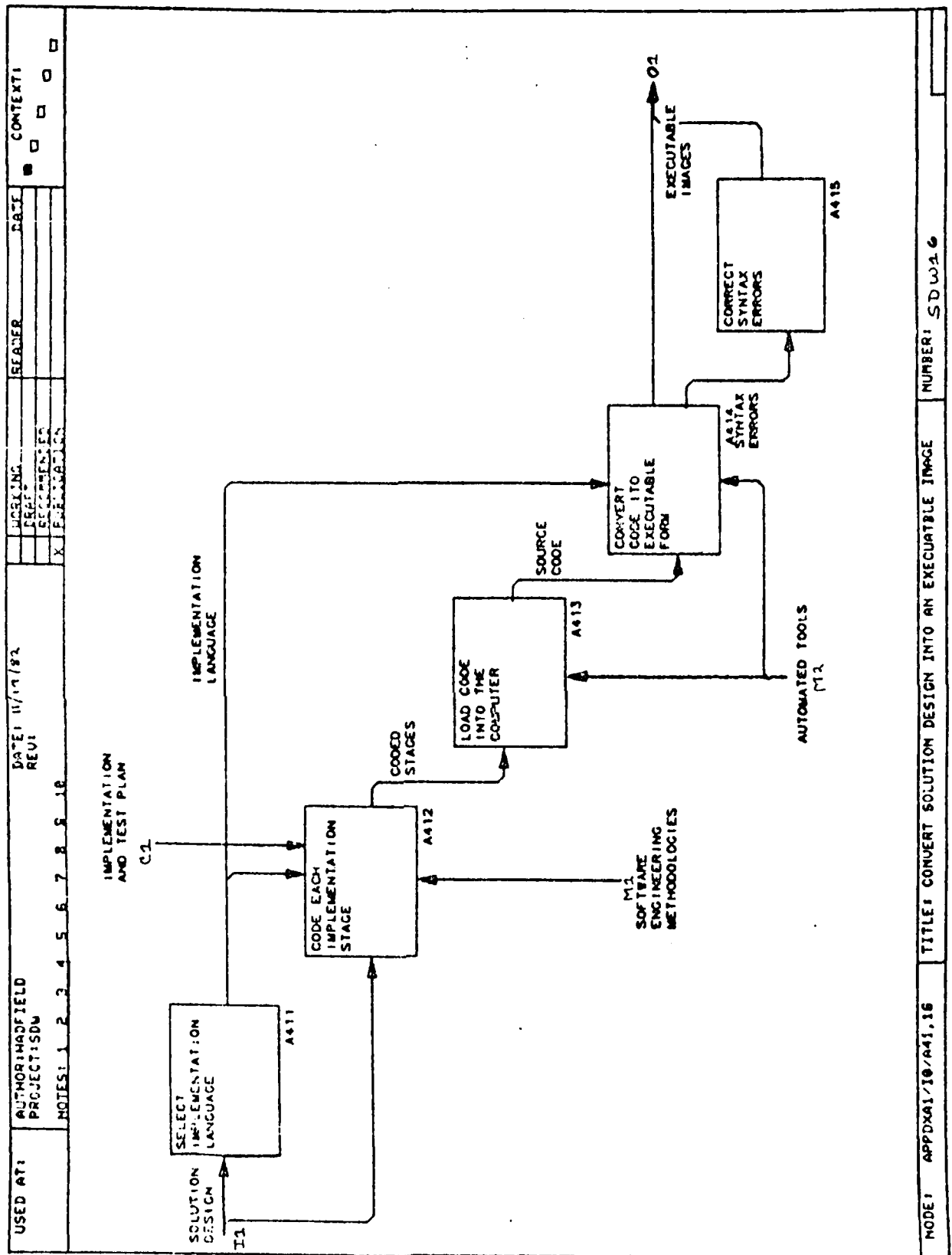
ACDE: APPDXA/10/A313.12

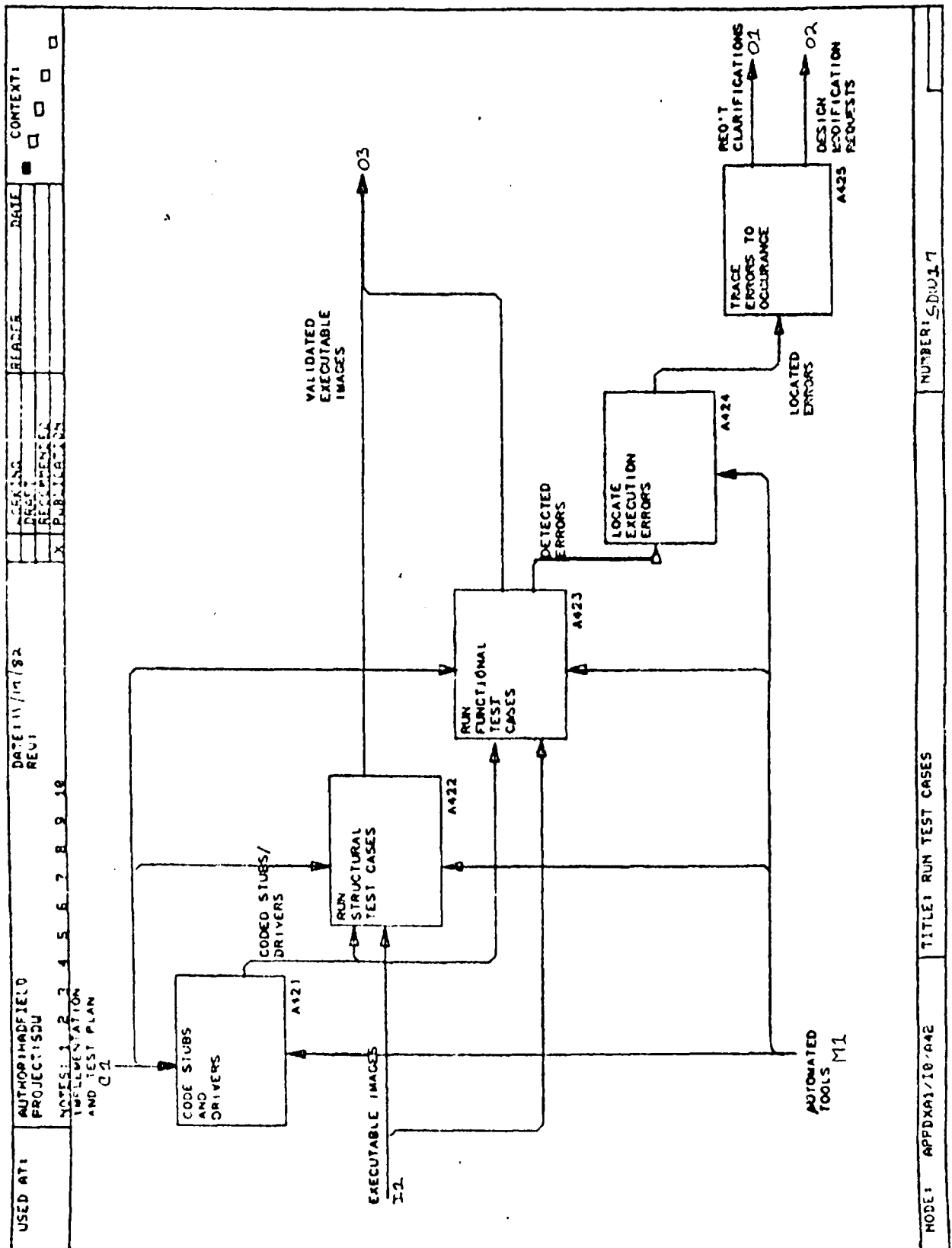
NUMBER: SDU.12



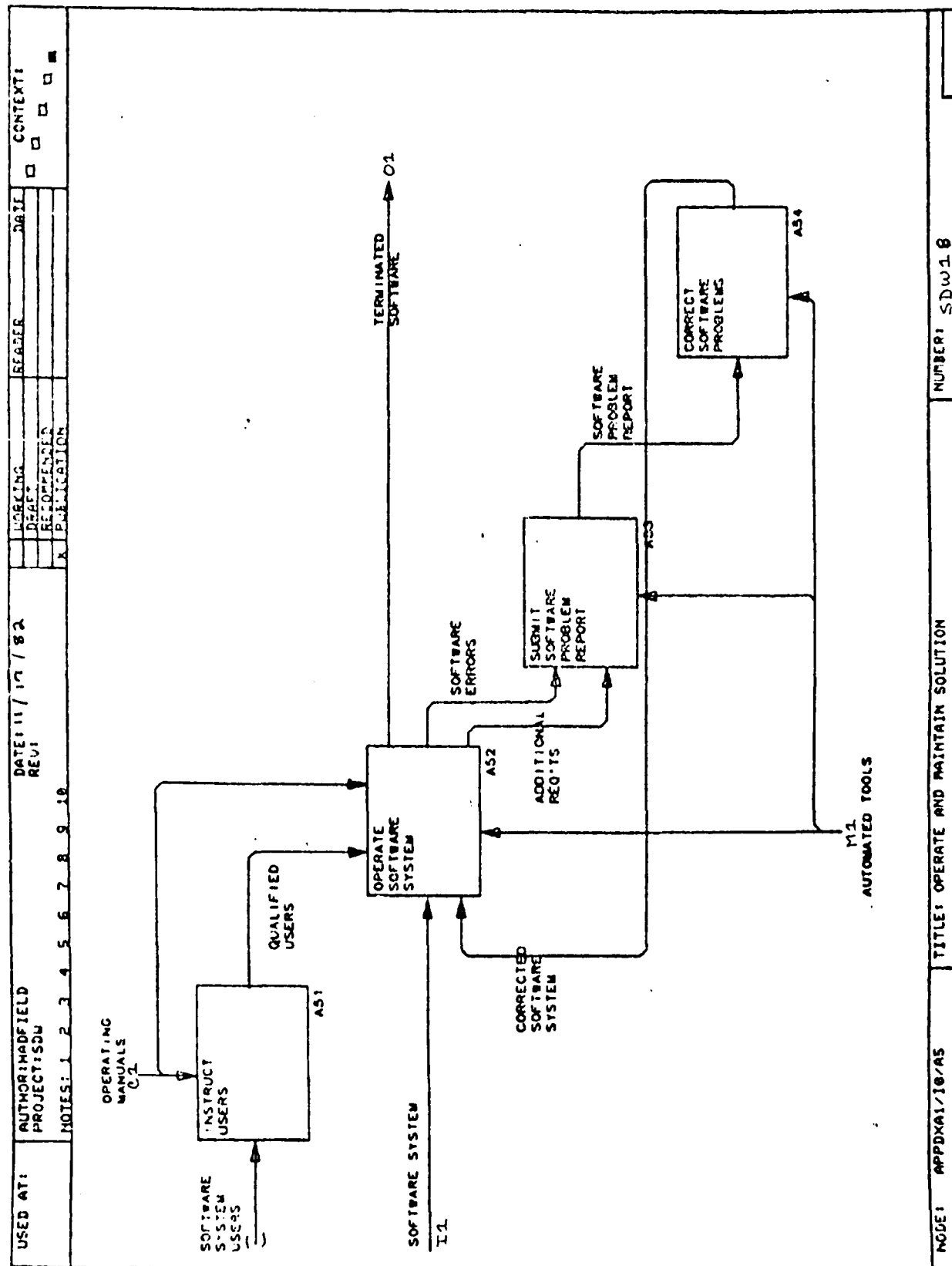












253

[illegible]

Appendix B: SDW Data Dictionary

### Data Dictionary for the Initial SDW Development

This data dictionary is provided to the reader to assist in the understanding of the models and concepts of this initial development of the AFIT Software Development Workbench. The data dictionary enumerates all data items important to the development effort and provides a brief description of each data item to insure that the reader understands its usage within the realm of the SDW development.

Data Item	Description
AIDES	An automated tool for constructing Structure Charts.
"As-Is" System Definition	A model of the current system being utilized to solve a particular problem
Augmentive Tools	Automated or interactive facilities used for testing and checking the specifications of requirements, design, and implementation documents.
AUTOIDEF	An automated tool for constructing IDEF0 and IDEF1 diagrams.
Backus Normal Form (BNF)	A means of formally specifying the syntax of a language. Also referred to as Backus-Naur Form.
Black-Box Test Plan	A testing strategy of inputs and expected outputs with no regard for the internal logic of the system.
BNF	Backus Normal Form.
CDM	Common Data Model
CIDS	Central ICAM Development System.
Code Formatters	Automated tools that reformat software

code to improve read-ability.

**Code Generators** Automated tools that assist in the translation of a software design into the actual code.

**Coding Errors** Errors in the software that occur and are detected during the Implementation Stage.

**Cognitive Tools** Automated and interactive facilities for developing software using common principles and practices of Software Engineering. Tools used to expand the conceptual capabilities of the developer.

**Common Data Model** A data base schema that stores the relationships between elements of other schemas.

**Compilers** Automated tools that convert source code into executable object code.

**Computer-Generated Requirements Diagrams** Two-dimensional graphical representations of the System Requirements.

**Configuration Managers** Tools that help to coordinate several versions of a particular system.

**Configuration Model** A model of the hardware, software, data base and other components of a system.

**Control Coupling** The level of coupling between design/code modules characterized by the passing of control variables.

**Data Coupling** The level of coupling between design/code modules characterized by the passing of data variables.

**Data Flow Analyzers** Automated tools that trace data items in a software system to detect data anomalies.

**Data Flow Tools** Tools that support system specification and design by analyzing the flows of data through system functions.

**Data Structure Tools** Tools that describe a system in terms of its data units and the necessary operations on those data units.

Debuggers	Automated tools that provide on-line tracing and incremental executions of software programs.
Defined Requirements	A version of the requirements for the system that check for completeness, accuracy, and consistency
Design Modification Request	The statement of a need to update and/or modify the Preliminary Design of a system.
Detailed Design	A statement of the algorithms and specific mechanisms that are to be used in the implementation of the system.
Detailed Design Document	A formal version of the Detailed Design.
DFDs	Data Flow Diagrams used for the stating of requirements in an easy to understand two-dimensional graphics format.
Draft Preliminary Designs	Intermediate versions of the Preliminary Designs
Draft Requirements	A intermediate version of the requirements for the system.
Draft Requirements Diagrams	The Draft Requirements stated in a two-dimensional graphics format.
EREVS	Extended Requirements Engineering and Validation System, an automated tool that allows for the specifying, testing, and modeling of concurrent/distributed system requirements.
Erroneous Algorithms	Algorithms that do not satisfy their specified functions.
Execution Profilers	Automated tools that traces how many times specific lines of code in a program are executed during a single run of the program.
Formal Mapping	A means of tracing among components of two different sets of entities.

Functional Cohesion	A design objective realized when all design modules are characterized by their own single function.
Functional Tool Group	A set of SDW component tools that have been classified together by the fact of similar function.
Hard Copy Graphics	The capability to produce two-dimensional graphics on paper.
HIPO	Hierarchical Input Process Output technique used for systems analysis and design.
Hosted Software System	A complete Software System that resides on the machine for which it was designed to execute.
ICAM	Integrated Computer-Aided Manufacturing.
ICAM/SEM	ICAM Systems Engineering Methodologies.
IDEF	ICAM DEFinition techniques.
IDEF0	IDEF for functional modelling.
IDEF1	IDEF for informational modelling.
IDEF2	IDEF for dynamic modelling.
IDSS	ICAM Decision Support System, A simulation tool.
Implementation	The stage of development during which the actual code for the software is developed.
Implementation Stages	Units of the incremental plan to implement the software.
Inconsistencies	Specifications within the system requirements that are incompatible.
Integration	The stage of software development when the coded software system is mated to the target machine.
Interactive Graphics Editors	Interactive automated tools that provide means to create and modify two-dimensional graphics images on a video display.



Interface Checkers	Automated tools that check the interfaces between software modules.
ISDS	Integrated Systems Development System
"Job Shop" Approach	An approach to the development of software development environments that utilizes a small set of highly integrated tools.
Linkers	Automated tools that merge and map together separate software units.
Machine-Readable Draft Requirements	The system requirements stated in a format that can be parsed by the computer.
Maintenance/Operation	The stage of software when the software is actually being used and changes to the existing system are made if required.
Mathematical Support Libraries	On-line libraries of software modules that provide numerical functions.
Modified Designs	Extended and updated versions of the Preliminary Designs.
N-Squared Charts	A graphical technique that utilizes matrices to illustrate interface between software modules.
Nassi-Sniederman Charts	A graphical technique used to illustrate the design of structured code. Can be used to prove if some code is structured.
Network Oriented Simulators	Simulation tools that view the system to be processed in terms of nodes and edges.
Notational Tools	Automated and interactive facilities that expand the developer notational powers.
Object Codes	The compiled and executable versions of a software system.
On-Line Software	Software that is currently operational on the target machine.
Operating Manuals	Guides used for instructing users on the operation of a particular software system.

Optimized Software System	An advanced version of the Software System that has been tuned for time performance and spacial usage.
Pattern Recognition	The capability of the computer to recognize patterns of vocal, visual or other analog inputs.
Performance Criteria	A parametric statement of acceptable performance ranges for a system, usually with respect to space and/or time.
Pre-Fabricated Software Description DB	A database and set of applications programs that provide for the cataloging and retrieval of already written software product descriptions.
Pre-Fabricated Software Product DB	A database containing the actual software products that are described by the Pre-Fabricated Software Description Database.
Preformance Monitors	Automated tools that trace the time spent in different areas of the code during execution.
Preliminary Design	A statement of the functional structure of the system usually done with HIPO or Structure Charts.
Preliminary Design Document	A formalized version of the Preliminary Design.
Pre-Fab Fuctions	Designed and coded software modules that fulfill a specified function.
Preliminary Design Validation Flag	A signal that the Preliminary Design has passed a set of tests design to calculate the designs validity.
Problem Understanding	A developed knowledge of the problem to be solved.
Process Oriented Simulators	Simulation tools that view the system to be processed as a series of inter-related processes.
Project DB	The data base that contains all of the documentation for a particular development effort.

R-net	A process flow diagram that describes system requirements for REVS/EREVS.
Requirements Clarification Request	A request to clarify an ambiguous requirement.
Requirements Document	The formal statement of the requirements for the proposed system.
Requirements Update Request	The statement of a need to modify or extend the requirements for the system.
Requirements Validated Flag	A signal that the stated requirements have passed the tests applied to them and thus are more likely correct.
Requirements Voids	Important aspects of the system that have not been specified.
REVS	Requirements Engineering and Validation System, used for specifying, testing, and modeling system requirements.
RSL	Requirements Specification Language, a language used by REVS/EREVS to state requirements in terms of entities, attributes, and relationships.
Qualified Users	A group of individuals that have been trained with and can use the software system.
SADT(TM)	Softech's Structured Analysis and Design Technique.
SDW	The AFIT Software Development Workbench
SDW Executive (SDWE)	The top-level module of the SDW that controls and coordinates all SDW functions.
Shared Data Base	A data base that is used by a variety of different applications.
Soft Copy Graphics	The capability to project two-dimensional graphics images on a video display.
Software Bottlenecks	Areas of the code where large amounts of time and/or space are being used up.
Software Engineering	The discipline that defines means to develop software using well specified methodologies.

Software Problem Report	A report that details an inadequacy found with the Software System as currently implemented.
Software System	The fully coded version of the system.
Source Code	The human-oriented listings of the software.
SSL	System Specification Language, the language used to initially describe a system to REVS/EREVS.
Structure Charts	A two-dimensional graphics technique used to describe the design of a system as a hierarchy of modules with specific interfaces.
Structure Chart Tools	Tools that automate the development of Structure Charts.
Structured Code	Software code that is constructed using a limited set of constructs that emphasize a single entrance point and a single exit point.
Structured English	A means of stating requirements and designs that uses a strictly formatted subset of the english language.
Symbolic Execution Tools	Automated tools that trace all functions of a program to all of their outputs and along all of their paths.
Syntax Errors	Attempts at code that is illegal within the constructs of the particular programming language.
Syntax-Directed Editors	Interactive automated tools that check the syntax of and compile the source code as it is entered into the computer.
System Concept	The realization of a problem to be solved by a Software System.
System Requirements	A statement of the functions that the system must be able to perform and the criteria within which it must perform the functions.
System/ Sub-System	Logical subsets of the software system.

## Components

System/  
Sub-System  
Component  
Specifications      Descriptions of the inputs, outputs, and  
functions of the System/Sub-System Components.

Target  
Environment  
Specifications      Specifications of the environment within which  
the software system will reside and operate.

Target  
Machine  
Emulators      Automated tools that make the host computer  
behave like the target computer.

Terminated  
Software  
System      The archived documentation of a Software  
System that is no longer needed or  
operational.

Test Case  
Generators      Automated (interactive) tools that produce  
sets of input data with which software  
systems are tested.

Test Coverage  
Analyzers      Automated tools that report which parts of  
a program were executed during a particular  
test run.

Test Plans      Strategies and test cases for use in the  
validating of the coded software.

Text Editors      Interactive automated tools that provide  
for the insertion and modification of text  
into files on the computer.

Textual  
Requirements      A statement of the System Requirements in  
an english language form.

"Tool Kit"  
Approach      An approach to developing software development  
environments that emphasizes the use of many  
independent component tools.

Unsatisfied  
Functions      Functions required by the Preliminary  
Design that have not been previously  
designed or coded.

Updated  
Requirements      A version of the system requirements that  
has been modified to allievate some problem.

Word  
Processing      An automated facility for developing  
textual manuscripts.

Appendix C: Specifications of Preliminary  
Design Modules

### Specification of Preliminary Design Modules

This appendix presents formatted specifications of the SDW Preliminary Design modules of the SDW Structural Model (3.5.2). Each module is described by the type of tool it is, the calling module, any subordinate calls, the inputs and outputs, a functional description, a comment block, and a special resolves entry. The resolves specification traces the module to the operation(s) of the SDW Functional Model (2.4) that it satisfies. The table below identifies all of the SDW Preliminary Design Modules.

#### SDW Preliminary Design Modules

<u>Number</u>	<u>Label</u>
3-1	Code Generators
3-2	Compilers
3-3	Configuration Managers
3-4	Consistency Checkers
3-5	Data Flow Analyzers
3-6	Debuggers
3-7	Dimension Checkers
3-8	Environmental Emulators
3-9	Execution Profilers
3-10	Functional Design Tools
3-11	Graphics Editors
3-12	Help Files
3-13	Information-Oriented Design Tools
3-14	Interface Checkers
3-15	Interface to Pre-Fab Software Description Data Base
3-16	Interface to Project Data Bases
3-17	Linkers
3-18	Loaders
3-19	Logic Path Analyzers
3-20	Performance Monitors
3-21	Planning Tools

3-22	Requirements Definition
	Tools
3-23	SDW Executive
3-24	Simulators
3-25	Source Code Formatters
3-26	Statistical Packages
3-27	Symbolic Execution Tools
3-28	Syntax-Directed Editors
3-29	Teach Routines
3-30	Test Case Generators
3-31	Test Result Comparators
3-32	Text Editors
3-33	Word Processors



=====

3-1 TYPE OF TOOL: Code Generators

RESOLVES: 1.4.3.1, 1.4.3.2

CALLED BY: Information-Oriented Design Tools,  
Functional Design Tools

CALLS: None

INPUTS: Some variety of Design Specification

OUTPUTS: Full or Partial High Order Language Source  
Code

FUNCTIONAL DESCRIPTION: A computer program that translates a design specification for a program into all or part of the actual source code for the program. Usually has two parts, (1) an analyzer that check the design specification and (2) the actual code generator.

COMMENTS: Code Generator capabilities are often embedded in Design Specification Tools

=====

3-2 TYPE OF TOOL: Compilers

RESOLVES: 1.4.3.4

CALLED BY: SDW Executive

CALLS: None

INPUTS: Source Codes

OUTPUTS: Object Codes, Error Diagnostics

FUNCTIONAL DESCRIPTION: Compiler programs convert source code versions of programs into executable object code. During the conversion process, the source code is checked for syntax and some semantical errors. Most all compilers are designed for a single programming language.

COMMENTS: The SDW must have compilers to support many different languages.

=====

=====

3-3 TYPE OF TOOL: Configuration Managers

RESOLVES: 1.4.2, 1.4.3.2, 1.4.4.1, 1.4.4.4, 1.4.5.2

CALLED BY: SDW Executive

CALLS: Interface to Project Data Bases

INPUTS: Software System Components

OUTPUTS: Version Data on Software System Components

FUNCTIONAL DESCRIPTION:

A configuration manager controls the different components and versions of a software development project.

=====

3-4 TYPE OF TOOL: Consistency Checkers

RESOLVES: 1.1.3, 1.2.2, 1.3.4

CALLED BY: Information-Oriented Design Tools, Functional Design Tools, and Requirements Definition Tools

CALLS: None

INPUTS: Machine-Readable Design Specifications,  
Machine-Readable Requirements Specifications

OUTPUTS: Listing of Voids and Inconsistencies

FUNCTIONAL DESCRIPTION: A computer program used to determine (1) if requirements and/or designs specified for computer programs are consistent with each other and their data bases and (2) if they are complete.

COMMENTS: Consistency Checkers are often embedded in Requirements Specification and Design tools.

=====

=====

3-5 TYPE OF TOOL: Data Flow Analyzers

RESOLVES: 1.4.4.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: High Order Language Source Code

OUTPUTS: Report of Data Flow Anomalies

FUNCTIONAL DESCRIPTION: A computer program that checks source code listings for data flow anomalies such as the referencing of a variable before it has been set.

COMMENTS: Data Flow Analyzers are often language specific.

=====

3-6 TYPE OF TOOL: Debuggers

RESOLVES: 1.4.4.3

CALLED BY: SDW Executive

CALLS: None

INPUTS: An Executable Version of a Program, Set of Specific User Commands

OUTPUTS: Listing of Variable Values, Location Values

FUNCTIONAL DESCRIPTION: A computer program that allows the user to control the execution of a program, monitor specific locations, change specific locations, check the sequence of program control and otherwise locate and correct errors as they occur during the execution of the program.

COMMENTS:

=====

=====

3-7 TYPE OF TOOL: Dimension Checkers (Units Checkers)

RESOLVES: 1.4.4.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: Source Code Listing, Dimensional Specification  
of the Physical Units associated with each Variable

OUTPUTS: Dimensional Inconsistency Report

FUNCTIONAL DESCRIPTION: Computer Programs that check  
assignment statements for dimensional consistency  
according to a user specified dimension for each variable  
referenced.

COMMENTS: Dimension Checkers are usually used for  
scientific programs and are language specific.

=====

3-8 TYPE OF TOOL: Environmental Emulators

RESOLVES: 1.4.4.3

CALLED BY: SDW Executive

CALLS: None

INPUTS: Specification of the Target Environment

OUTPUTS: An Emulated Environment

FUNCTIONAL DESCRIPTION: A computer program used to  
permit testing operational programs on a host computer.  
The operational programs run under emulated conditions  
as if they were operating within the real-time control  
program of a machine to which all of the devices constituting  
the ultimate system are attached.

COMMENTS:

=====

=====

3-9 TYPE OF TOOL: Execution Profilers

RESOLVES: 1.2.1, 1.2.3

CALLED BY: SDW Executive, Compilers

CALLS: None

INPUTS: High Order Language Source Code

OUTPUTS: A Source Code Listing that specifies the  
number of times each line of code was executed.

FUNCTIONAL DESCRIPTION: A computer program that records  
and reports how many times each line of source code in  
a program was executed during a specific run of that  
program.

COMMENTS: Execution Profilers are sometimes language  
specific and may require special options to be invoded  
during compilation.

=====

3-10 TYPE OF TOOL: Functional Design Tools

RESOLVES: 1.2.1.1, 1.2.1.2, 1.2.1.3, 1.2.2, 1.2.3,  
1.3.2, 1.3.3, 1.3.4

CALLED BY: SDW Executive

CALLS: Simulators, Consistency Checkers, Code Generators

INPUTS: Specification of Design in a Design  
Specification Language or in terms of 2-D Graphics

OUTPUTS: Specification of a program's Functional Design  
in Consistent and Unambiguous terms

FUNCTIONAL DESCRIPTION: An interactive computer program  
that assists the software developer in specifying the  
functional design of a software system. This specification  
is usually in terms of a recognized Software Engineering  
Methodology that focuses on the specification of functions  
as components of the software program.

COMMENTS: Examples of Functional Design Tools support  
methodologies such as HIPO, SADT, HOS, etc.

=====

3-11 TYPE OF TOOL: Graphics Editor

RESOLVES: 1.1.2.1, 1.1.2.4

CALLED BY: SDW Executive

CALLS: None

INPUTS: Specific Set of Terminal Commands,  
Existing Files of 2-D Graphics

OUTPUTS: Files of 2-D Graphics

FUNCTIONAL DESCRIPTION: A computer program that provides capabilities to create and update files of 2-dimensional graphics images. Primitive graphics editors will just display the graphics on a video screen and not provide the means to save images in files.

COMMENTS: Graphics editors are often dependent on very specific hardware

=====

3-12 TYPE OF TOOL: Help Files

RESOLVES: 1.7.2, 1.7.3

CALLED BY: SDW Executive

CALLS: None

INPUTS: Help Requests

OUTPUTS: Textual Displays of information and instructions to assist the user

FUNCTIONAL DESCRIPTION: A collection of files that information to assist the user in operating the SDW and its component tools. Help files are indexed so that requests can be answered quickly and efficiently.

COMMENTS:

=====

=====

3-13 TYPE OF TOOL: Information-Oriented Design Tools

RESOLVES: 1.2.1, 1.2.2, 1.2.3

CALLED BY: SDW Executive

CALLS: Code Generators, Consistency Checkers,  
Interface to the Project Data Base

INPUTS: Data Structure Specifications & Relationships

OUTPUTS: Informational Models and Designs

FUNCTIONAL DESCRIPTION: These computer programs use graphical and textual specifications of the data structures used in developments to produce data models and software designs based on the data structures.

COMMENTS: These tools support Software Engineering methodologies such as Jackson's Method, Warnier-Orr technique, etc...

=====

3-14 TYPE OF TOOL: Interface Checkers

RESOLVES: 1.3.4, 1.4.4.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: Source Code, Design Specifications

OUTPUTS: Interface Correctness Reports

FUNCTIONAL DESCRIPTION: Computer programs that check the number, order, type, and range of parameters that are passed between design and code modules.

COMMENTS:

=====

=====

3-15 TYPE OF TOOL: Interface to Pre-Fab Software Description  
Data Base

RESOLVES: 1.3.1

CALLED BY: SDW Executive

CALLS: Pre-Fab Software Description Data Base Manager

INPUTS: Set of Keywords, Description of a Software Unit

OUTPUTS: Description of a Software Unit

FUNCTIONAL DESCRIPTION: An applications program that provides the user with the ability to locate existing software units that may satisfy some of his required functions. The program uses a set of keywords to locate descriptions of candidate software units. This program also facilitates the entering of descriptions for new software units.

=====

3-16 TYPE OF TOOL: Interface to the Project Data Base

RESOLVES: 1.1.2, 1.1.6, 1.2.1, 1.2.3, 1.2.4, 1.3.2  
1.3.3, 1.4.3.2, 1.4.3.3, 1.4.4.4, 1.7.1

CALLED BY: Configuration Manager, Functional Design  
Tools, Information-Oriented Design Tools, Requirements  
Definition Tools

CALLS: Project Data Base Manager

INPUTS: Queries on the Project Data Bases, Project  
Development Data

OUTPUTS: Various Views of the Project Data Bases

FUNCTIONAL DESCRIPTION: An applications program that provides means for the user to interact with the Project Data Bases. In these data bases are stored all of the development data with separate schemas for each stage and another schema that preserves the mapping between the schemas of the different stages.



=====

3-17 TYPE OF TOOL: Linkers

RESOLVES: 1.4.4.1

CALLED BY: SDW Executive

CALLS: None

INPUTS: Separately Compiled Software Units

OUTPUTS: An Executable Image

FUNCTIONAL DESCRIPTION: A program that assembles and maps together separately compiled software units.

COMMENTS: This facility is usually included as a standard utility of the resident operating system

=====

3-18 TYPE OF TOOL: Loaders (Relocatable Loader)

RESOLVES: 1.4.4.1

CALLED BY: SDW Executive

CALLS: None

INPUTS: An Executable Image

OUTPUTS: A Loaded Executable Image

FUNCTIONAL DESCRIPTION: A computer program that enables external references of symbols among different assemblies as well as the assignment of absolute addresses to relocatable strings of code. This program provides diagnostics on assembly overlap, unsatisfied external references, and multiple defined external symbols.

COMMENTS:

=====

=====

3-19 TYPE OF TOOL: Logic Path Analyzers

RESOLVES: 1.4.4.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: Executable software

OUTPUTS: Path Analysis Report

FUNCTIONAL DESCRIPTION: A computer program that checks all logical paths within a program to determine if the paths are executable and what conditions must be satisfied for that path to be executed.

COMMENTS:

=====

3-20 TYPE OF TOOL: Performance Monitors

RESOLVES: 1.4.5.1

CALLED BY: SDW Executive

CALLS: None

INPUTS: A Software Program

OUTPUTS: Report on the Time Spent in each Module

FUNCTIONAL DESCRIPTION: A computer program that inserts extra code into a software program that records the amount of time spent in each area of the code during the execution of the software. Used for the detection of software areas where gains may be had by optimization.

COMMENTS:

=====

**UNCLASSIFIED**

AN INTERACTIVE AND AUTOMATED SOFTWARE DEVELOPMENT  
ENVIRONMENT(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING S M HADFIELD DEC 82  
AFIT/GCS/EE/82D-17 F/G 9/2

474

NL

END

FOLMEC



**MICROCOPY RESOLUTION TEST CHART**  
**NATIONAL BUREAU OF STANDARDS-1963-A**

=====

3-21 TYPE OF TOOL: Planning Tools

RESOLVES: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6

CALLED BY: SDW Executive

CALLS: None

INPUTS: Specification of the Development Effort

OUTPUTS: A Schedule for the Development Effort

FUNCTIONAL DESCRIPTION: An automated tool that aids in the creation and maintenance of software development plans and schedules.

COMMENTS:

=====

3.22 TYPE OF TOOL: Requirements Definition Tools

RESOLVES: 1.1.1, 1.1.2, 1.1.6

CALLED BY: SDW Executive

CALLS: Simulators, Consistency Checkers, Interface to the Project Data Bases

INPUTS: Specifications of Software Requirements

OUTPUTS: Various Test Reports on the Stated Requirements, Various Illustrations of the Requirements

FUNCTIONAL DESCRIPTION: Computer programs that accept a specification of system/software requirements (usually in a tool-specific specification language) and assemble them in a data base. Then, these requirements may be simulated, updated, checked for consistency/completeness, or used to produce different representations of the requirements.

COMMENTS: These tools often have Consistency checkers and Simulators built into them.

=====

=====

3-23 TYPE OF TOOL: SDW Executive

RESOLVES: 1.

CALLED BY: None

CALLS: All SDW Component Modules

INPUTS: User commands

OUTPUTS: Prompts, Diagnostics

FUNCTIONAL DESCRIPTION: An operating system level program that controls the operations of the SDW and all of its components.

COMMENTS:

=====

3-24 TYPE OF TOOL: Simulators

RESOLVES: 1.1.5, 1.2.2.4

CALLED BY: SDW Executive, Functional Design Tools, Requirements Definition Tools

CALLS: None

INPUTS: Simulation Model

OUTPUTS: Requested Reports from Simulation Model

FUNCTIONAL DESCRIPTION: Computer programs that are used to estimate how particular systems, stated as simulation models, will perform under stated conditions over time. Simulations may be of software systems, target environments, other hardware components, etc...

COMMENTS: Simulators may be stand-alone, or embedded in other tools.

=====

=====

3-25 TYPE OF TOOL: Source Code Formatters

RESOLVES: 1.4.3.2, 1.4.3.3, 1.4.4.4

CALLED BY: SDW Executive

CALLS: None

INPUTS: Source Code Listings of Software

OUTPUTS: Formatted Source Code Listings

FUNCTIONAL DESCRIPTION: A computer program that insures that the source code listings of software are well formatted for readability and understandability.

COMMENTS:

=====

3.26 TYPE OF TOOL: Statistical Packages

RESOLVES: 1.1.1.1, 1.1.1.5

CALLED BY: SDW Executive

CALLS: None

INPUTS: Applications Programs, Empirical Data

OUTPUTS: Statistical Reports

FUNCTIONAL DESCRIPTION: Computer programs that are used to statistically analyze data in ways prescribed by specific applications programs.

COMMENTS:

=====

=====

3-27 TYPE OF TOOL: Symbolic Execution Tools

RESOLVES: 1.4.4.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: Source Codes of Software

OUTPUTS: Trace Reports

FUNCTIONAL DESCRIPTION: Computer programs used to trace coded software along all possible paths or partial paths and evaluate the execution along those paths symbolically.

COMMENTS:

=====

3-28 TYPE OF TOOL: Syntax-Directed Editors

RESOLVES: 1.1.2.2, 1.1.2.3, 1.1.4, 1.1.6, 1.2.1, 1.2.2.3  
1.2.3, 1.3.2, 1.3.3, 1.4.3, 1.4.4.4, 1.4.5.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: BNF Description of a Programming Language,  
User-Entered Source Code

OUTPUTS: Source and Object Codes of Software

FUNCTIONAL DESCRIPTION: A text editor that checks the syntax and compiles source code as it is entered. The editor does not allow syntactically-erroneous source code to be entered.

COMMENTS: Syntax-Directed Editors are extend to also accept constraints from previous development stages to insure that entered text is both syntactically correct and valid against the previously stated constraints.

=====



=====

3-29 TYPE OF TOOL: Teach Routines

RESOLVES: 1.7.2

CALLED BY: SDW Executive and all SDW components

CALLS: None

INPUTS: Teach Requests (Specific or General)

OUTPUTS: Scripts for Operating the SDW and Components

FUNCTIONAL DESCRIPTION: Automated aids that provide on-line teaching of the operation of the SDW and the SDW components.

COMMENTS: Separate teach routines are provided for each of the SDW components.

=====

3-30 TYPE OF TOOL: Test Case Generators

RESOLVES: 1.4.4.3, 1.6

CALLED BY: SDW Executive

CALLS: None

INPUTS: Software Source Codes, Data Constraints

OUTPUTS: Input Test Cases

FUNCTIONAL DESCRIPTION: Computer programs that analyze software codes and data constraints and automatically produce input test cases used to test the software code.

COMMENTS:

=====

=====

3-31 TYPE OF TOOL: Test Result Comparators

RESOLVES: 1.4.4.3, 1.4.4.5

CALLED BY: SDW Executive

CALLS: None

INPUTS: Anticipated Results, Actual Results

OUTPUTS: Descripencies Between Actual & Anticipated  
Results

FUNCTIONAL DESCRIPTION: Automated tools that compare  
the actual output of test runs to the anticipated output  
of test runs and report any descripencies.

COMMENTS:

=====

3-32 TYPE OF TOOL: Text Editors

RESOLVES: 1.1.1.4, 1.1.2.2, 1.1.6, 1.2.1, 1.2.3, 1.3.2,  
1.3.3, 1.4.3.2, 1.4.4.4, 1.4.5.2

CALLED BY: SDW Executive

CALLS: None

INPUTS: Text, User Commands

OUTPUTS: Text Files

FUNCTIONAL DESCRIPTION: Computer programs that provide  
for the interactive insertion and modification of text  
which is saved in text files.

COMMENTS: Text Editors are often used to create &  
modify source codes and in conjunction with word  
processors.

=====

=====

3-33 TYPE OF TOOL: Word Processors

RESOLVES: 1.1.1, 1.1.1.2, 1.2.1, 1.3.3, 1.7.1

CALLED BY: SDW Executive

CALLS: None

INPUTS: Text files, User commands

OUTPUTS: Formalized Documentation

FUNCTIONAL DESCRIPTION: Computer programs that convert textual inputs into formalized documentation in accordance with specific commands.

COMMENTS:

=====

Appendix D:

Detailed Requirements Definition for the  
Software Development Workbench Executive

## Detailed Requirements Definition for the Software Development Workbench Executive

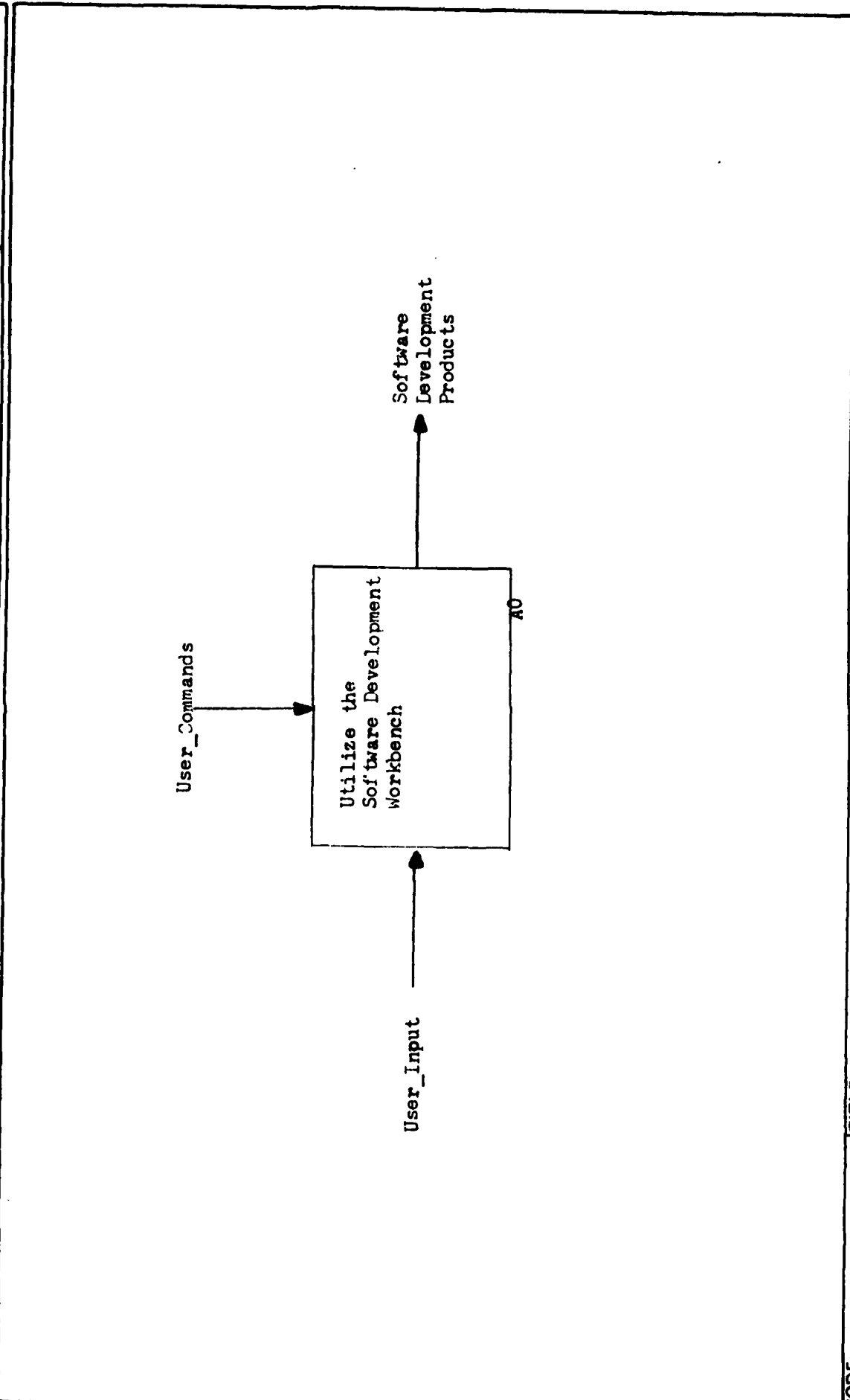
The Detailed Requirements Definition for the Software Development Workbench Executive (SDWE) defines the specific requirements for the SDWE which is the integrating interface and controller for the Software Development Workbench. The model defined in this appendix depicts the updated requirements for the SDWE current to publishing of this thesis document. The model uses the Structured Analysis and Design Technique (SADT). This technique is defined and its use justified in section 4.3.1 of this document.

### The Detailed Requirements Definition for the SDWE

- A-0 Utilize the SDW
  - A0 Utilize the SDW
    - A4 Execute the User's Command
      - A41 Provide a Functional Tool Group
      - A42 Provide Help Facilities
      - A43 Provide SDW Utility Functions
      - A44 Access the Pre-Fab Software Description Data Base

ST252

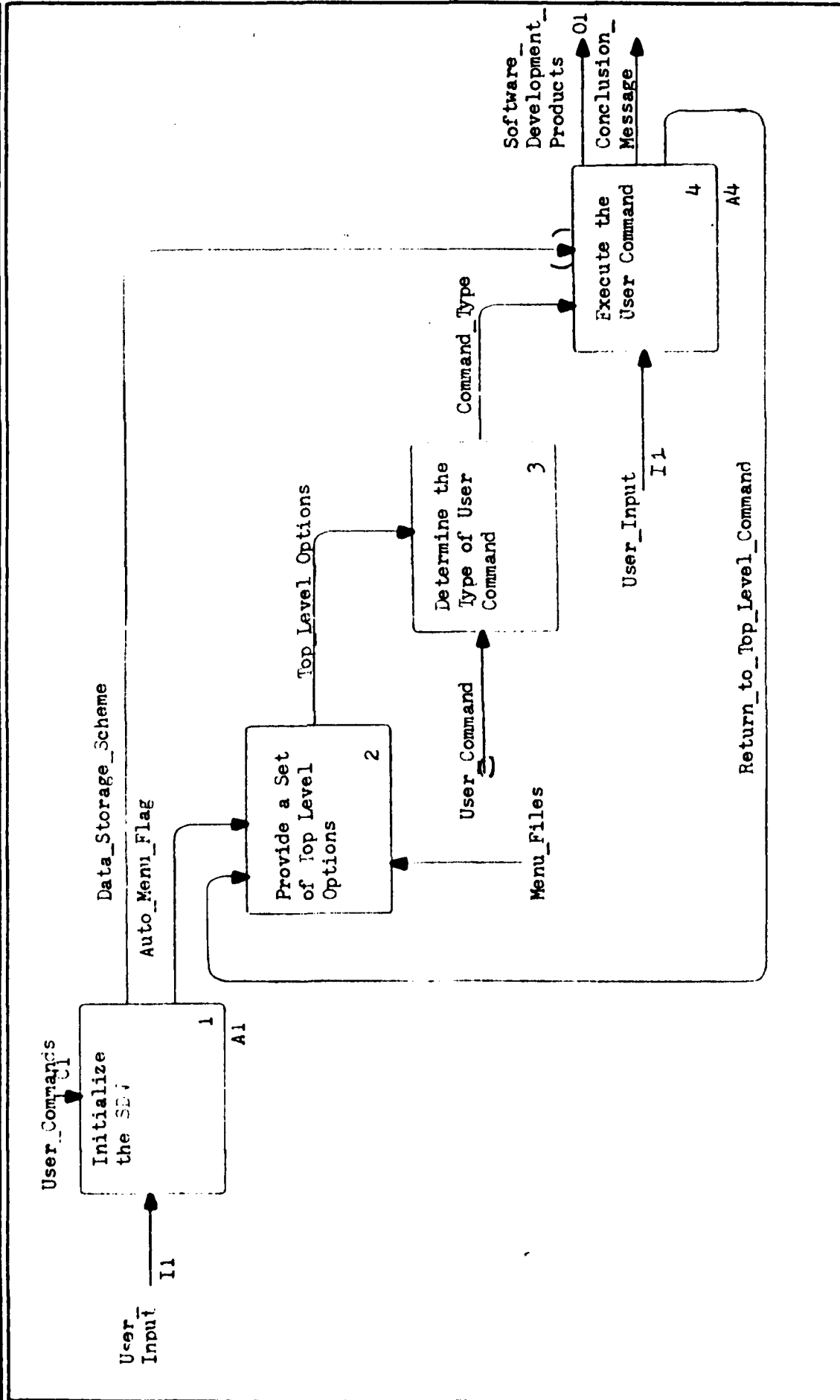
USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SL-2 Requirements	REV:	DRAFT			
NOTES: 1 2 3 4 5 6 7 8 9 10			RECOMMENDED			
			PUBLICATION			



NODE: A-0	TITLE: Utilize the SDW	NUMBER: SDW1
-----------	------------------------	--------------

SI252

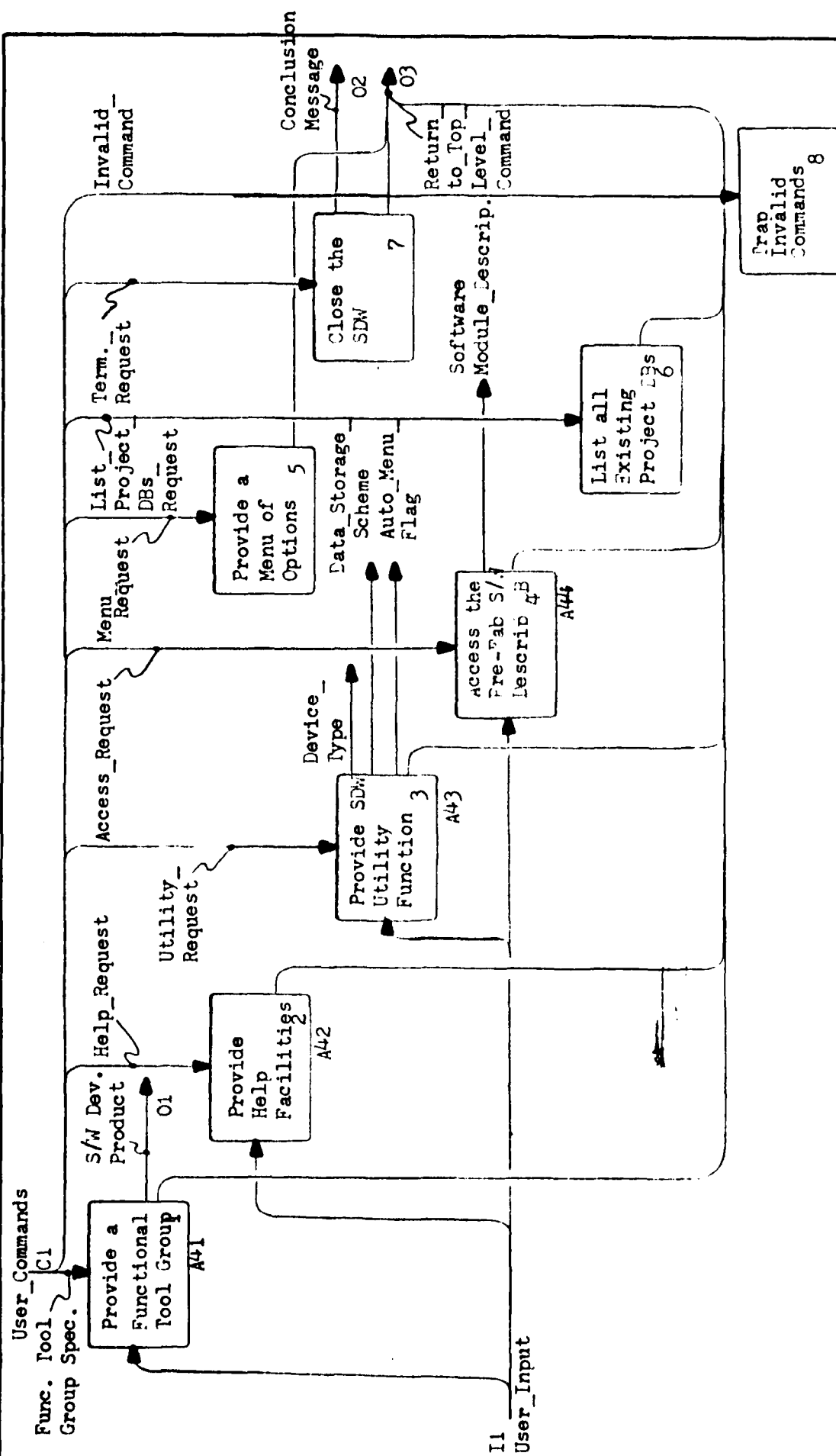
USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SEW Requirements	REV:	DRAFT			
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			
			PUBLICATION			



MODE: A0	TITLE: Utilize the Software Development Workbench	NUMBER: SI252
----------	---	---------------

ST252

USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
PROJECT: SDWE Requirements	REV:		DRAFT			
NOTES: 1 2 3 4 5 6 7 8 9 10			RECOMMENDED			
			PUBLICATION			A0



NODE: 41

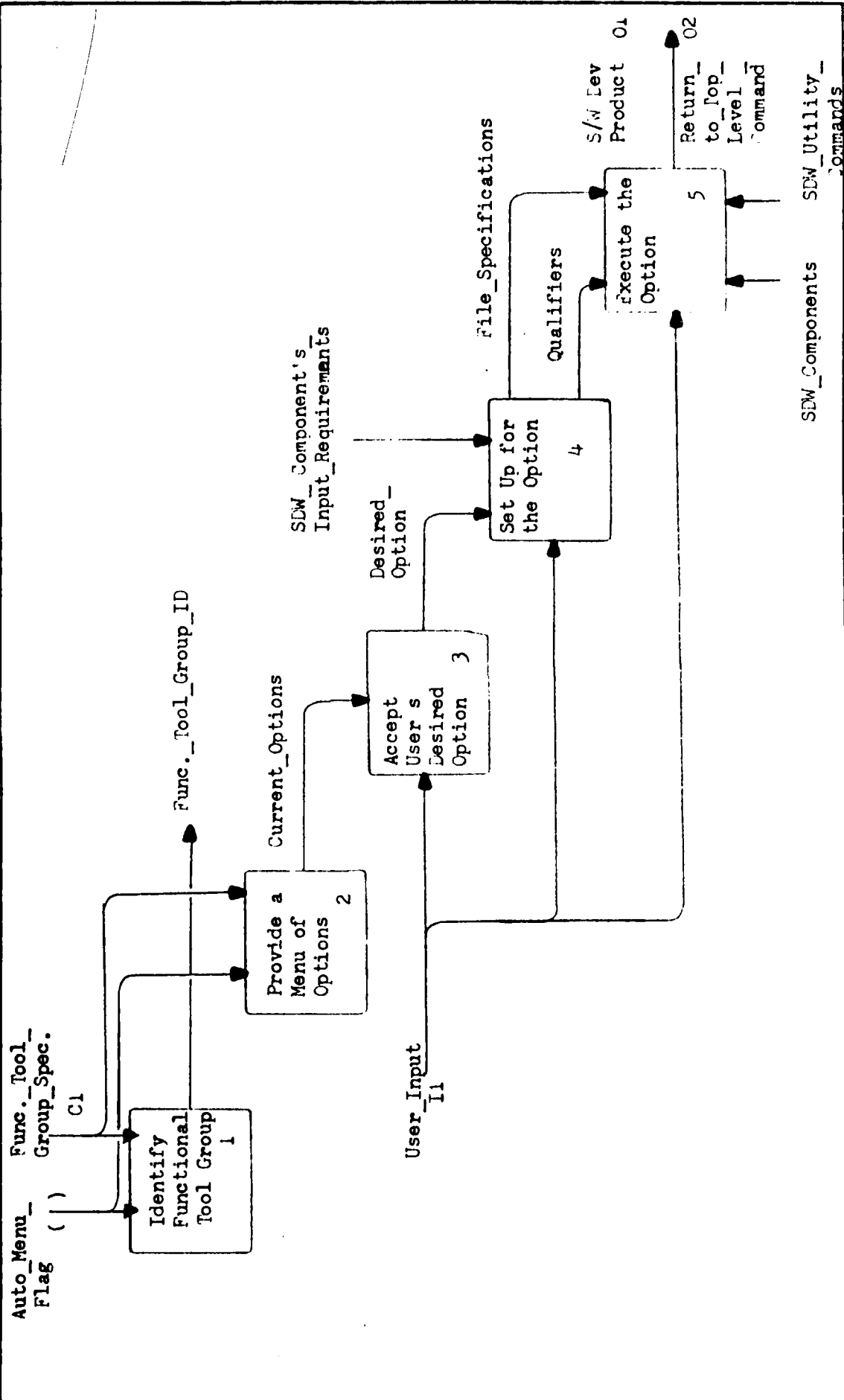
TITLE: Execute the User's Command

NUMBER: SDWE3



ST252

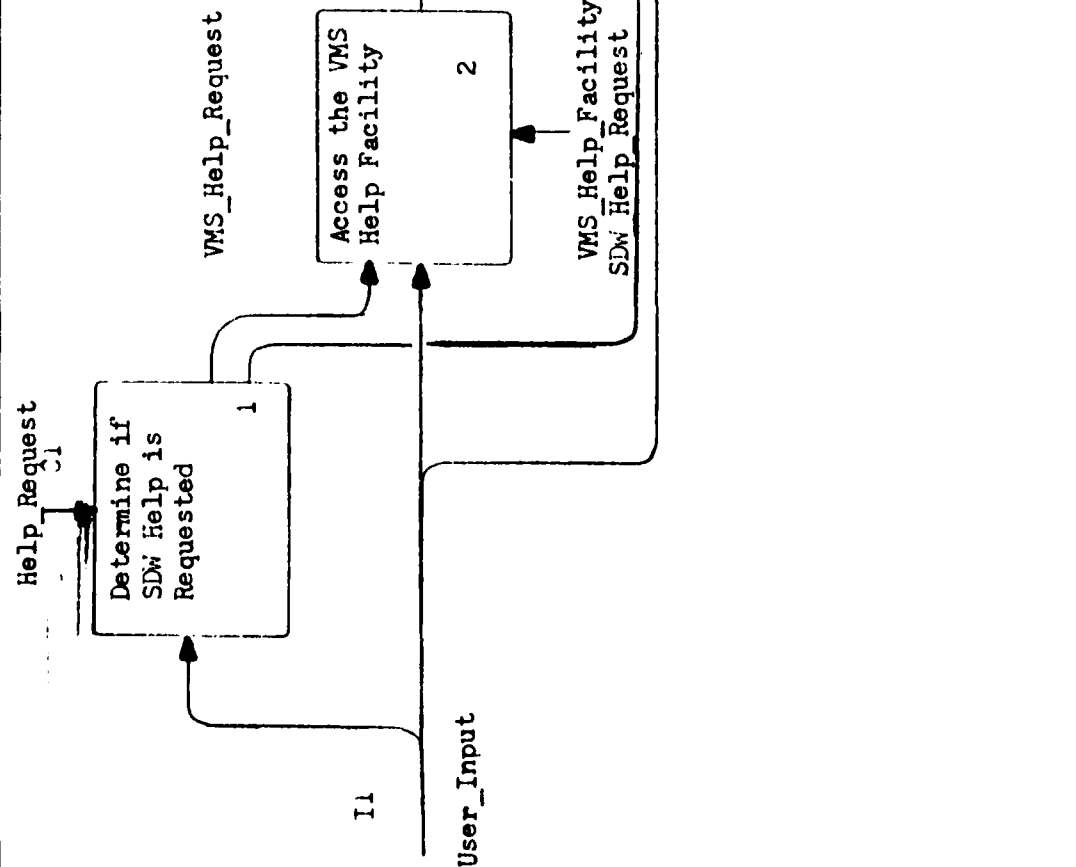
USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
PROJECT: SDWE Requirements		REV:	DRAFT			<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
NOTES: 1 2 3 4 5 6 7 8 9 10			RECOMMENDED			A4
			PUBLICATION			



NODE: A4.	TITLE: Provide a Functional Tool Group	NUMBER: SDWE4
-----------	--	---------------

ST252

USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SDW Requirements	REV:	DRAFT			<input type="checkbox"/>
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			<input type="checkbox"/>
			PUBLICATION			<input type="checkbox"/>
						A4



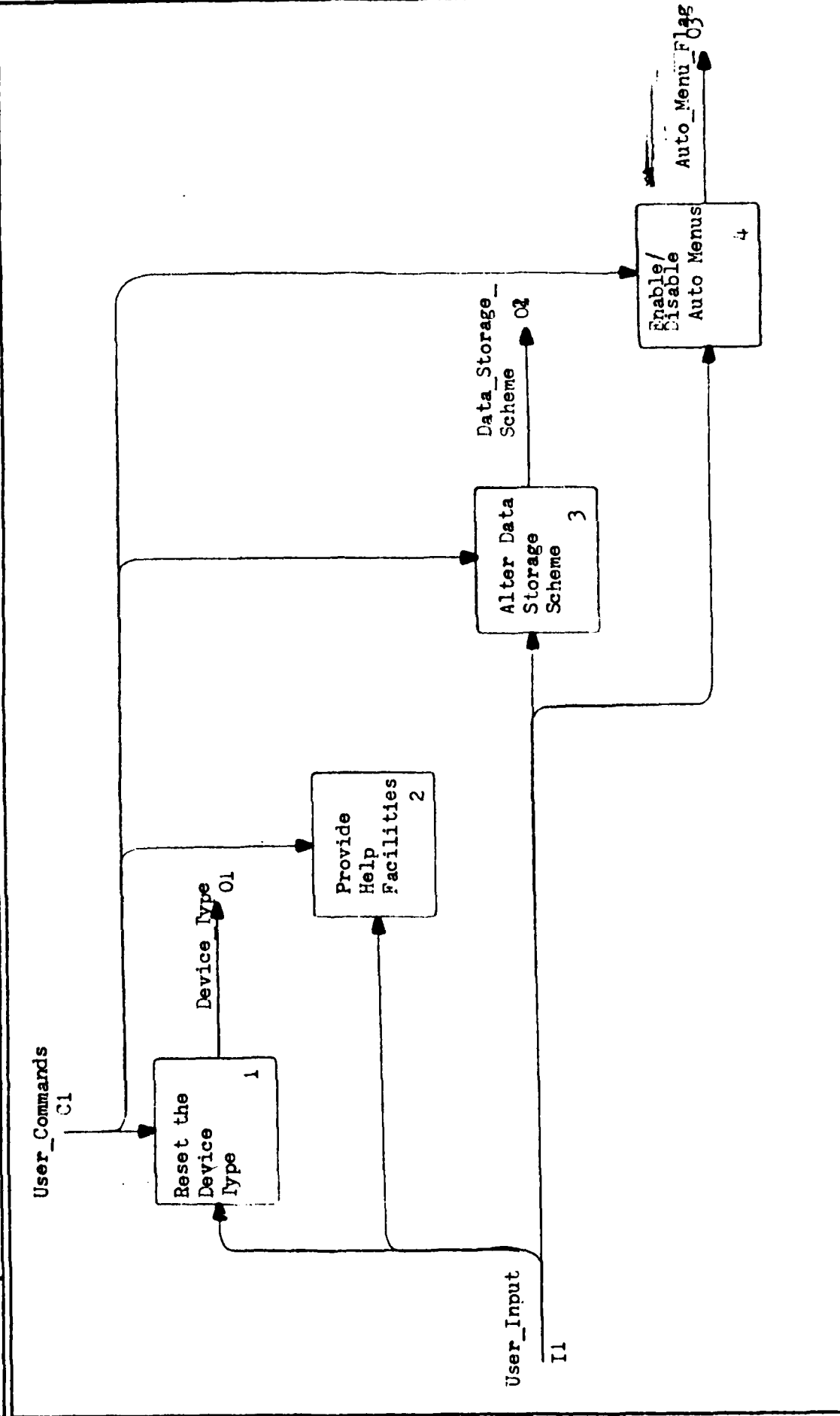
NODE: A42

TITLE: Provide Help Facilities

NUMBER: SDWE5

SI252

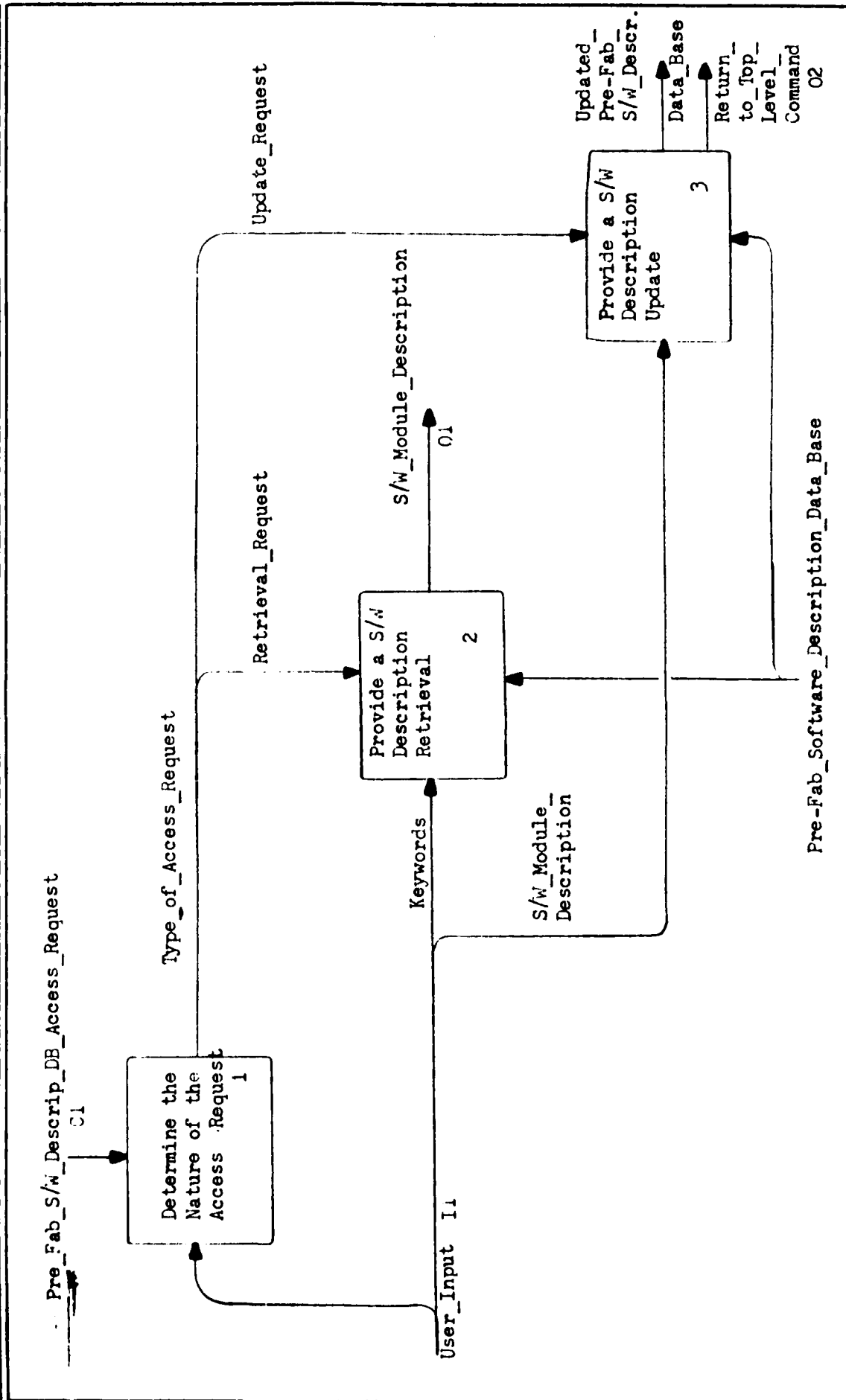
USED AT:	AUTHOR: Lt. Steven Hadfield	DATE: 1 Oct 82	WORKING	READER	DATE	CONTEXT:
	PROJECT: SDWE Requirements	REV:	DRAFT			<input type="checkbox"/>
	NOTES: 1 2 3 4 5 6 7 8 9 10		RECOMMENDED			<input type="checkbox"/>
			PUBLICATION			<input type="checkbox"/>
						A4



NODE: A43	TITLE: Provide SDW Utility Functions	NUMBER: SDWE6
-----------	--------------------------------------	---------------

ST252

USED AT:	AUTHOR: Lt. Steven Hadfield										DATE: 1 Oct 82		WORKING	READER	DATE	CONTEXT:	
	PROJECT: SDNE Requirements										REV:						
NOTES: 1 2 3 4 5 6 7 8 9 10												DRAFT			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
												RECOMMENDED				<input type="checkbox"/>	<input type="checkbox"/>
												PUBLICATION				<input type="checkbox"/>	<input type="checkbox"/>
																44	



NODE: A44	TITLE: Access the Pre-Fab Software Description Data Base	NUMBER: SDW42
-----------	--	---------------

Appendix E:  
Preliminary Design for the  
Software Development Workbench Executive

Preliminary Design for the  
Software Development Workbench Executive

The Preliminary Design Specification for the Software Development Workbench Executive (SDWE) defines the structural design model for the SDWE which is the integrating interface and controller for the Software Development Workbench (SDW). The model defined in this appendix depicts the updated Preliminary Design of the SDWE current to the publishing of this thesis document. The model uses the Structure Chart technique (Ref 90:141). This technique is defined and its use justified in section 4.3.2 of this document.

The Preliminary Design for the SDWE  
-----

<u>Figure</u> -----	<u>title</u> -----
5-1	Top Level Diagram
5-2	Provide Compilers
5-3	Provide Comparators
5-4	Provide Dynamic Analysis Tools
5-5	Provide Debuggers
5-6	Provide Design Tools
5-7	Provide Editors
5-8	Provide Graphics Editors
5-9	Provide Linkers
5-10	Access Pre-Fab S/W Description DB
5-11	Provide Performance Monitors
5-12	Provide Requirements Definition Tools
5-13	Provide Static Analysis Tools
5-14	Provide Simulation Tools
5-15	Provide Test Case Generators
5-16	Provide Word Processors

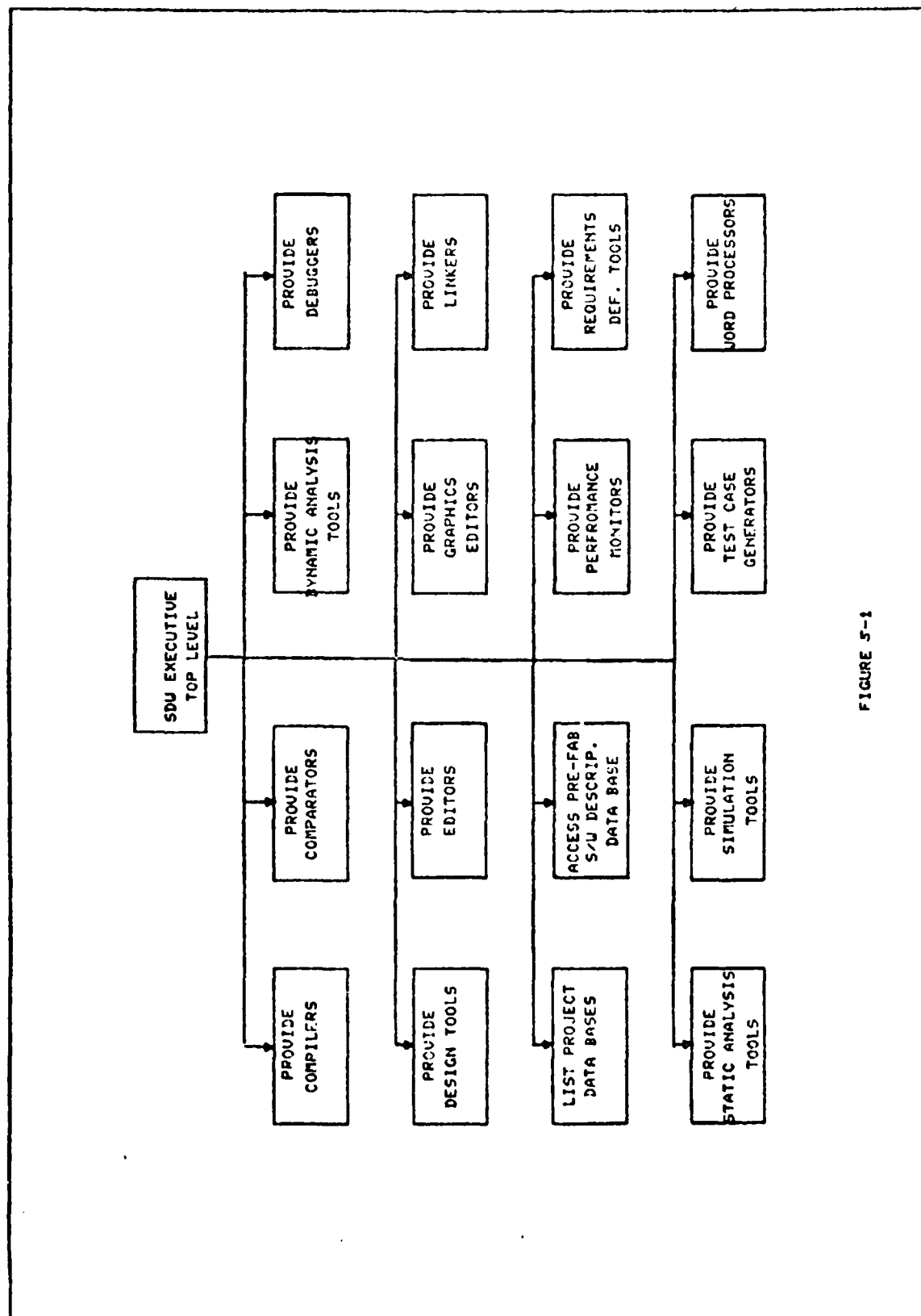


FIGURE 5-1

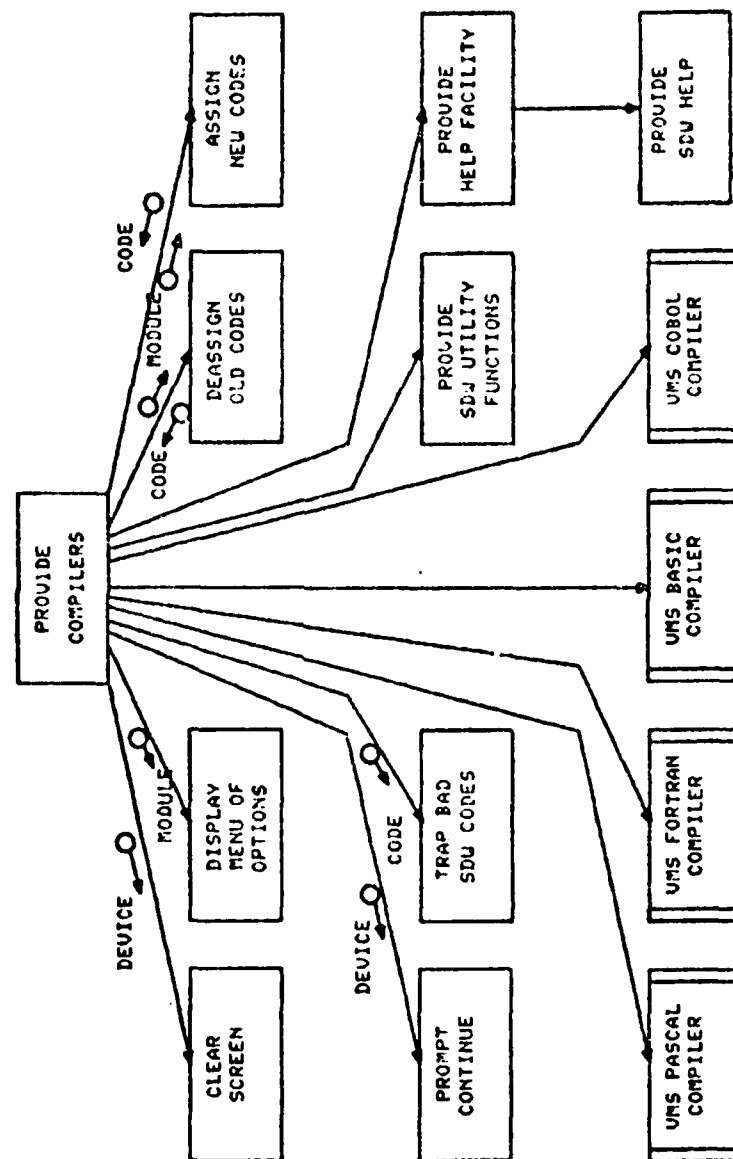


FIGURE 5-2



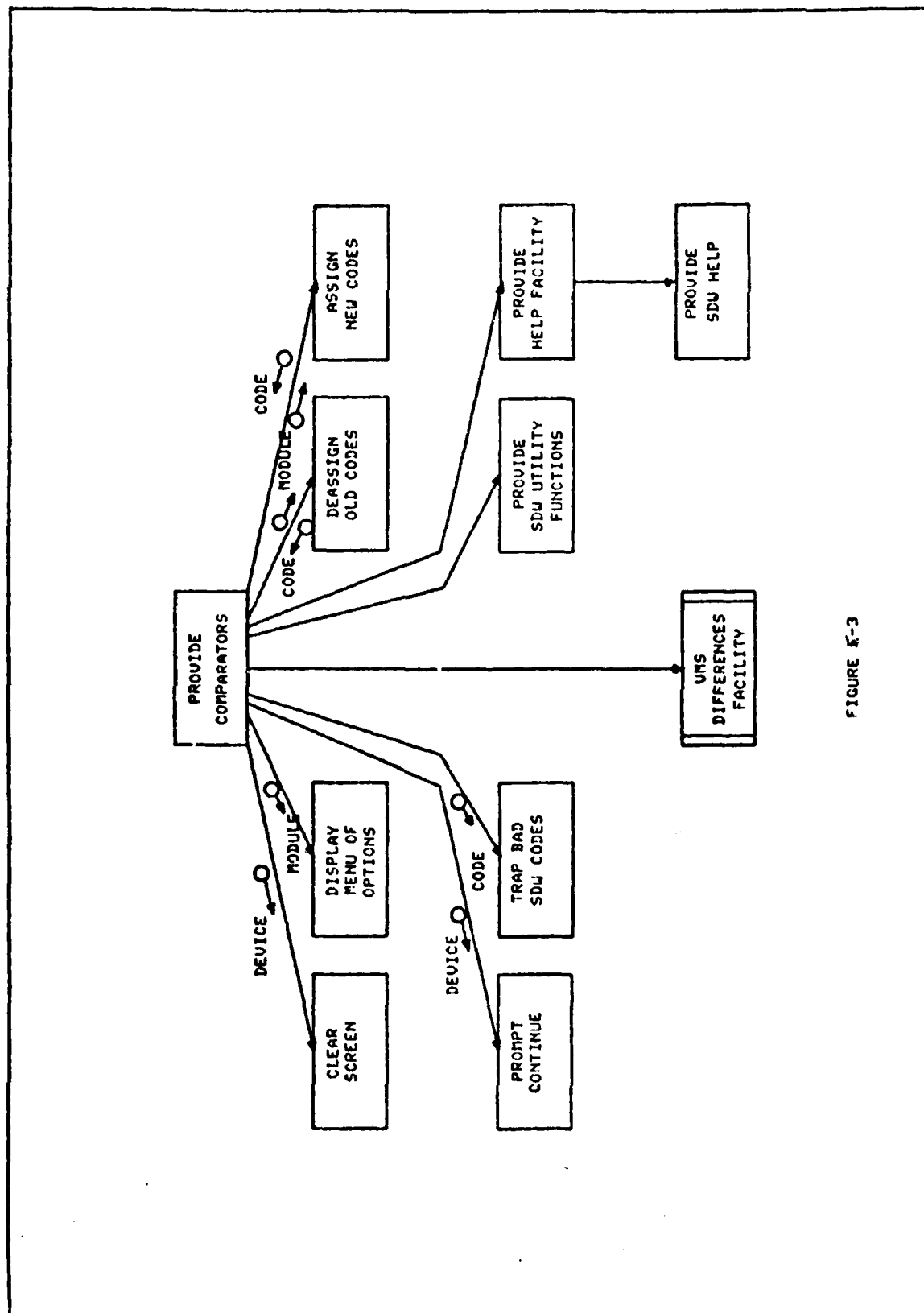


FIGURE K-3

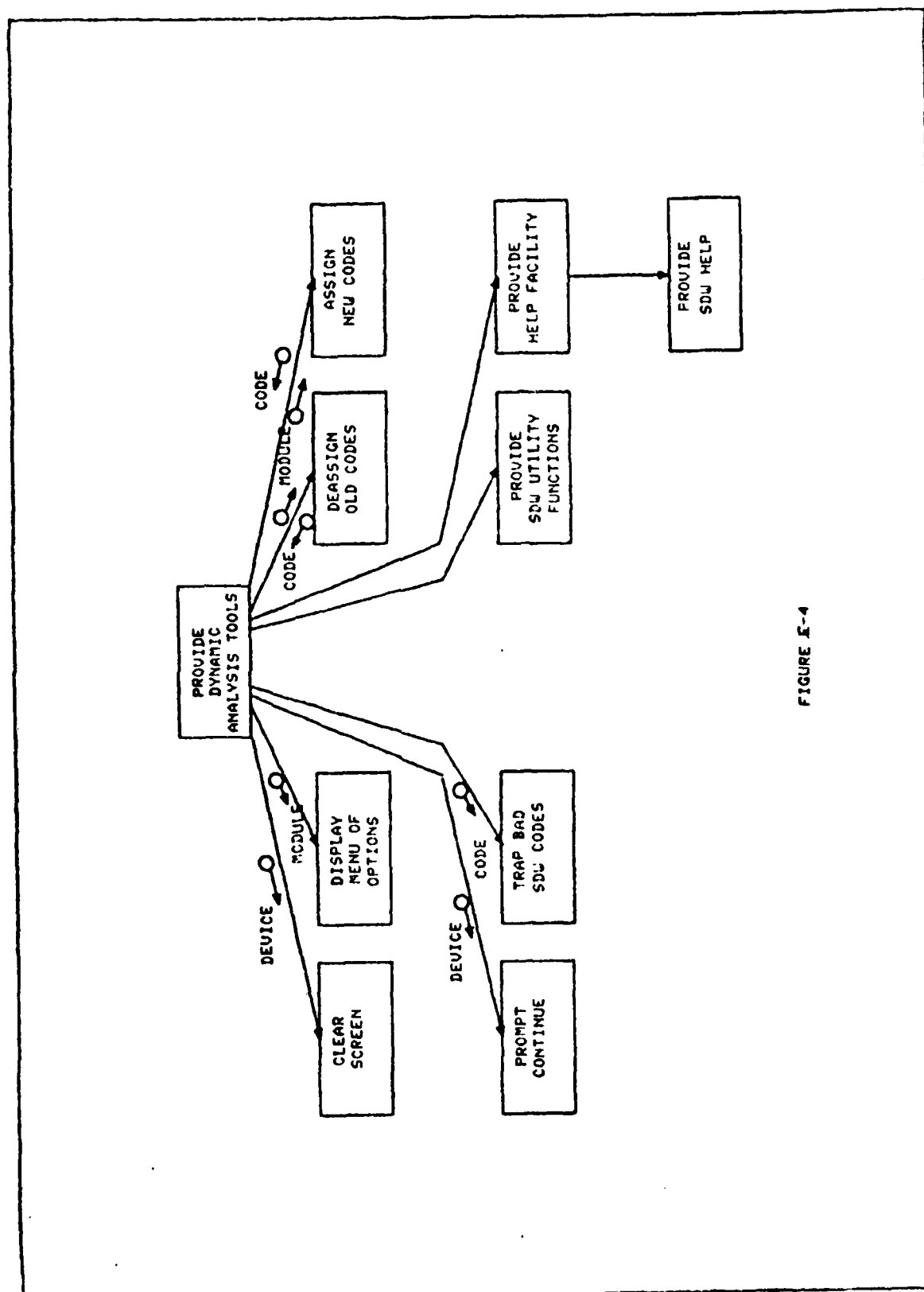


FIGURE E-4

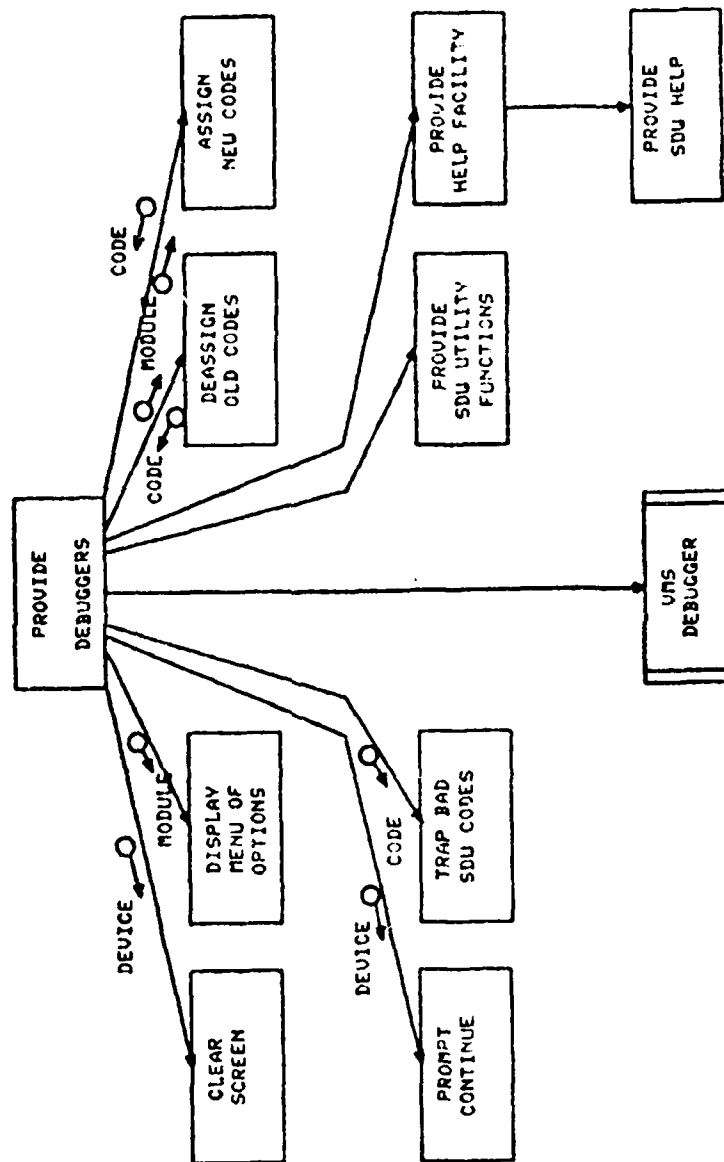


FIGURE 3-5

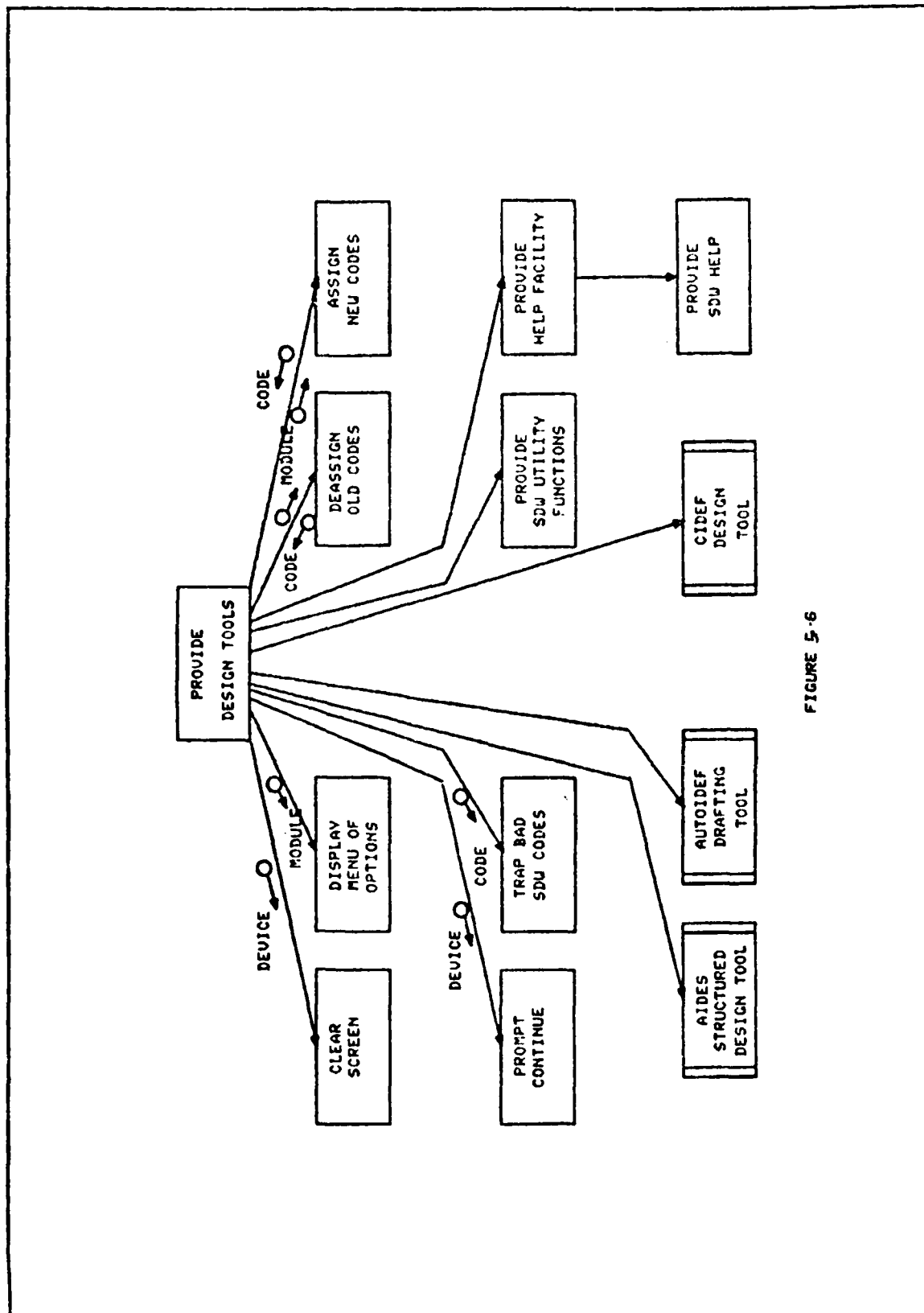


FIGURE 5-6

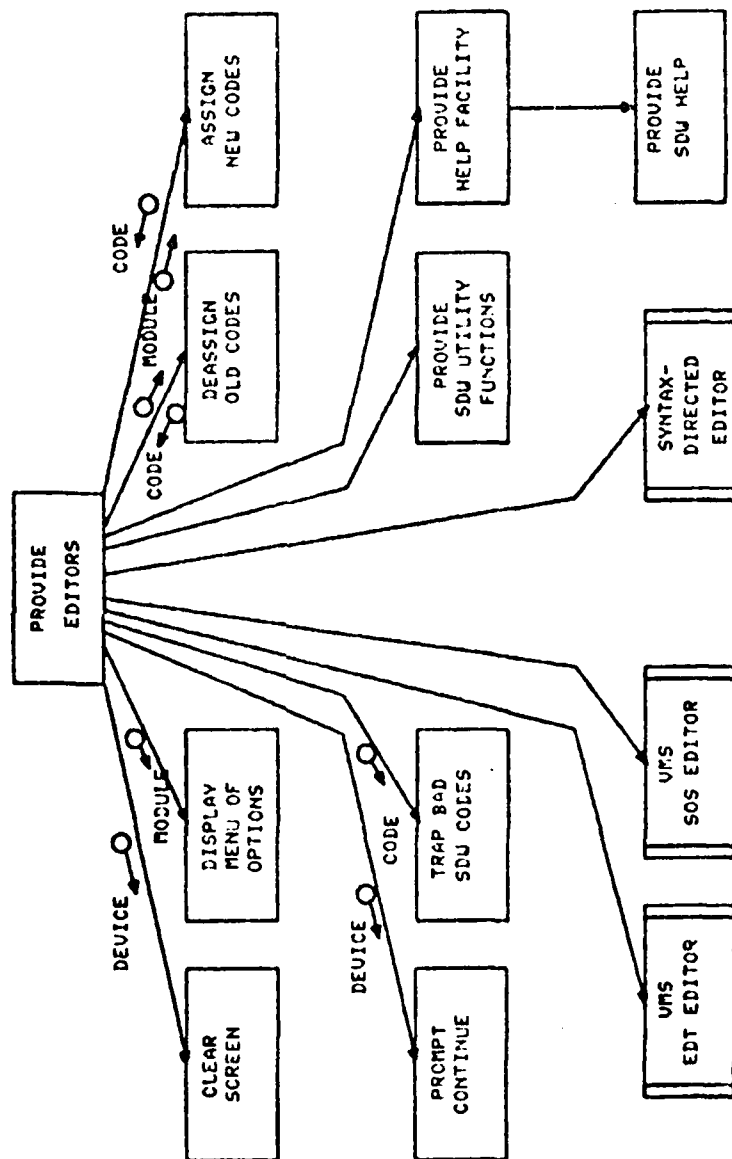
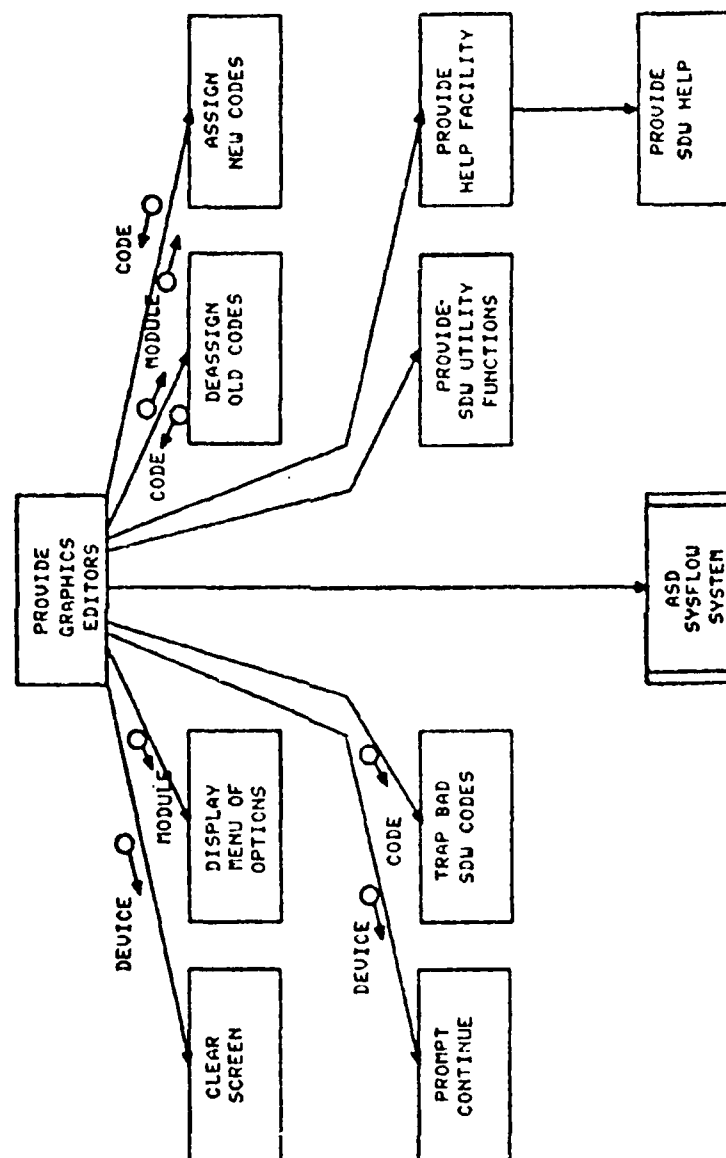


FIGURE 5-7



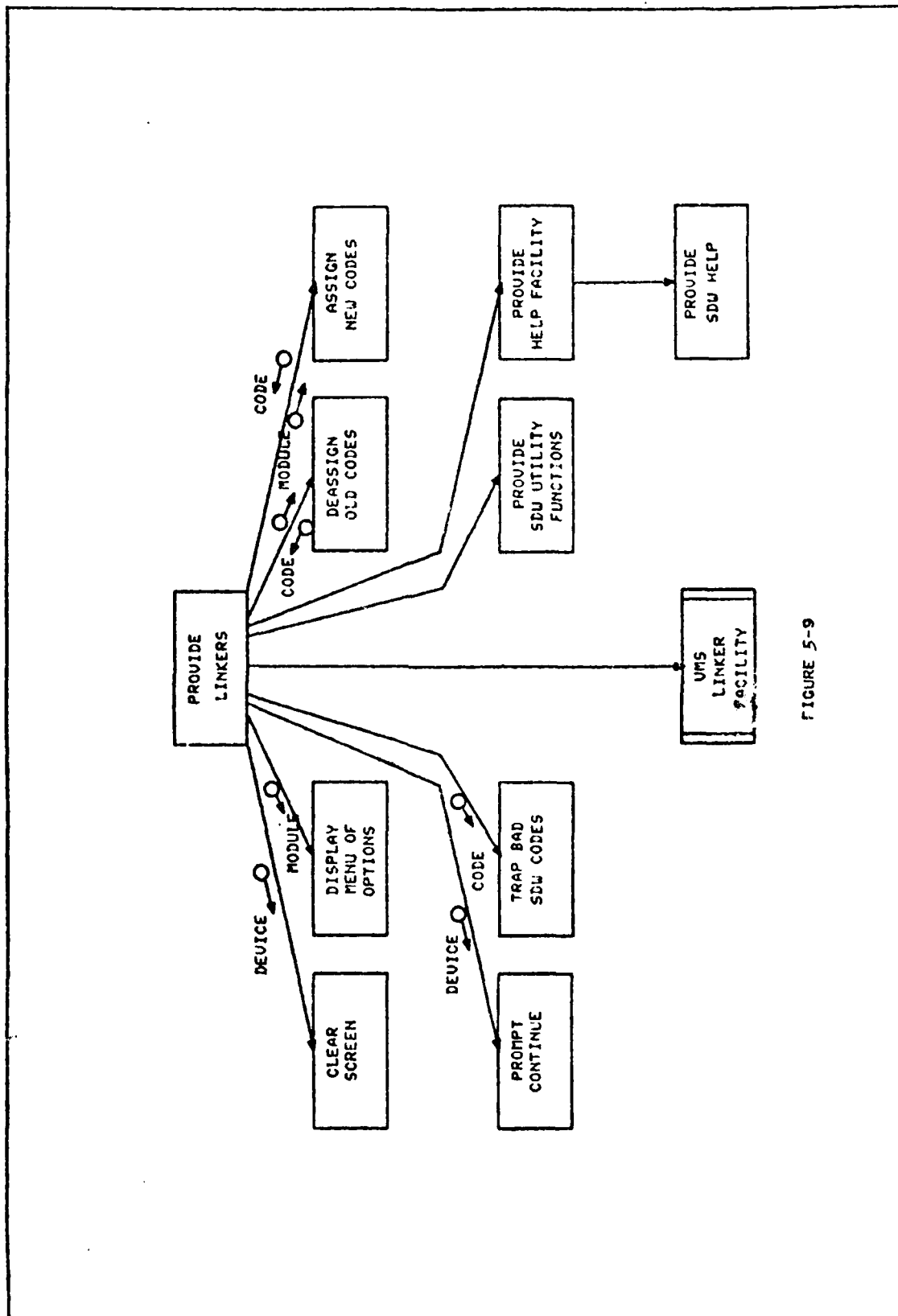


FIGURE 5-9

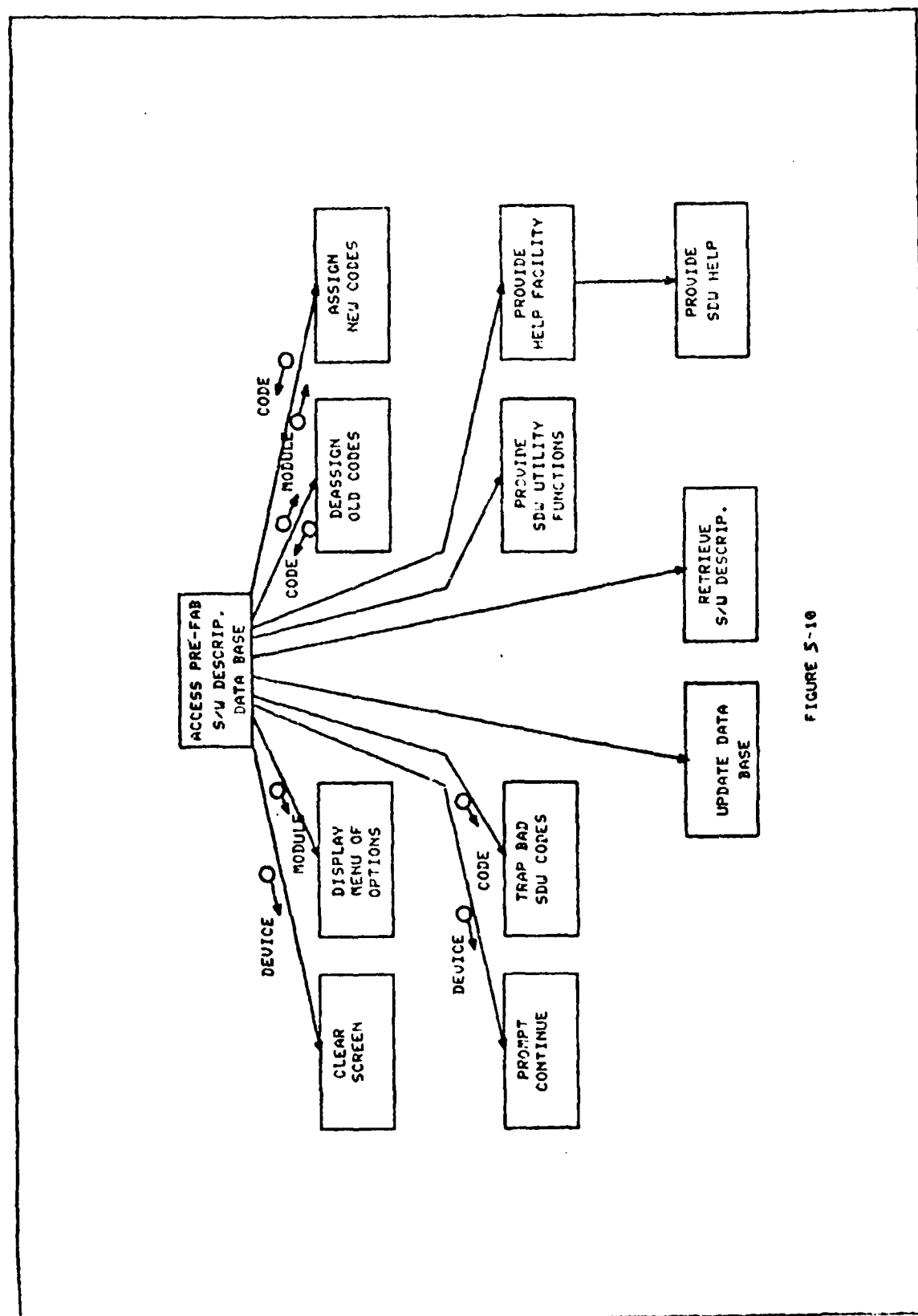


FIGURE 5-10



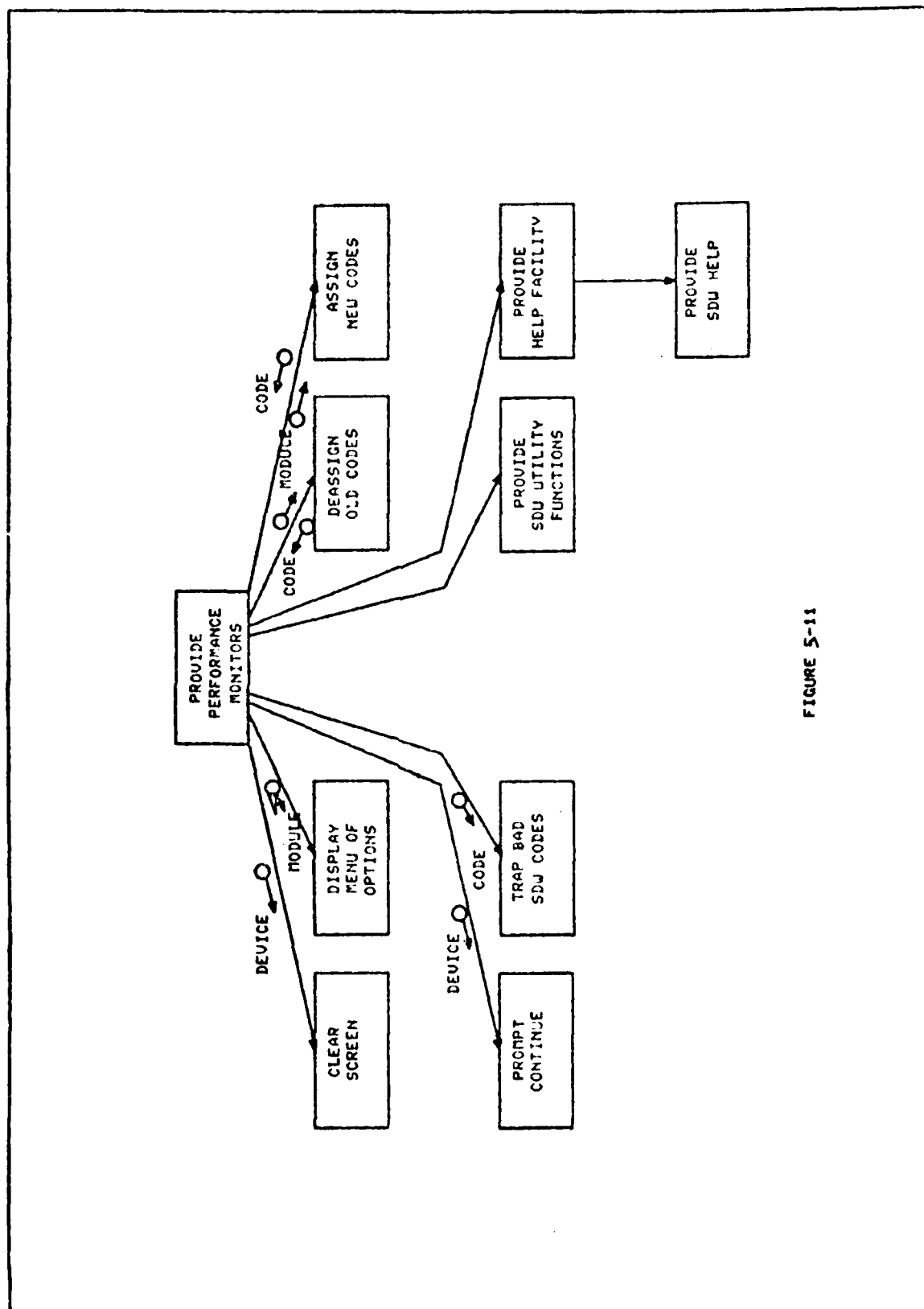


FIGURE 5-11

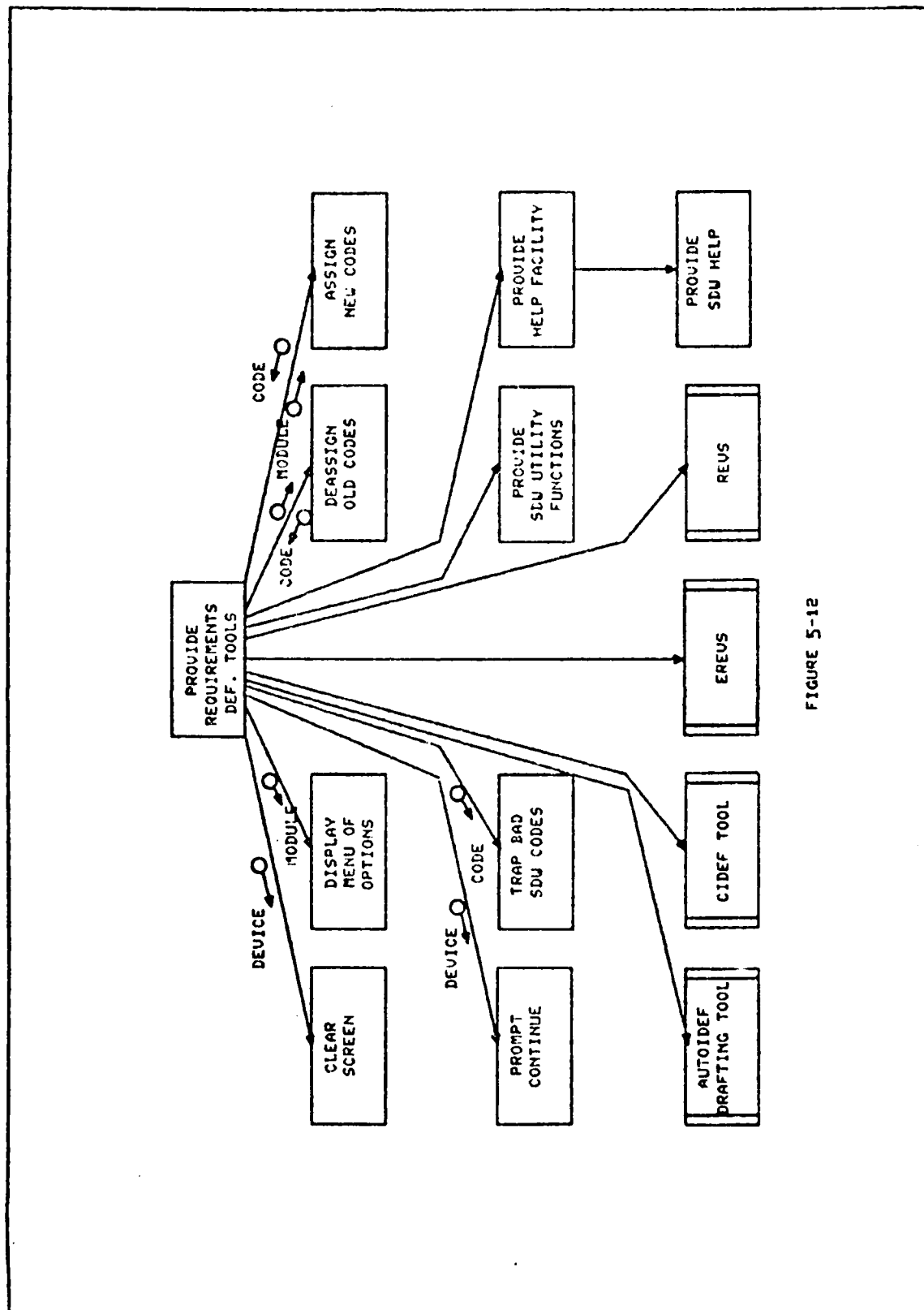


FIGURE 5-12

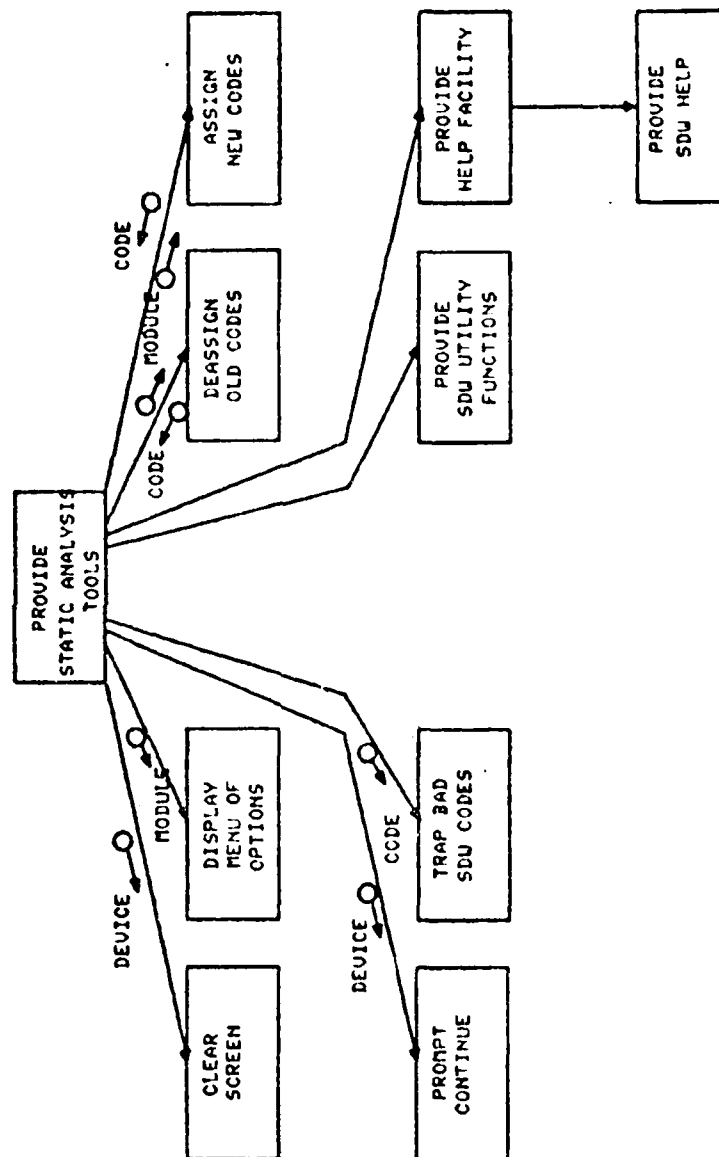


FIGURE 5-13

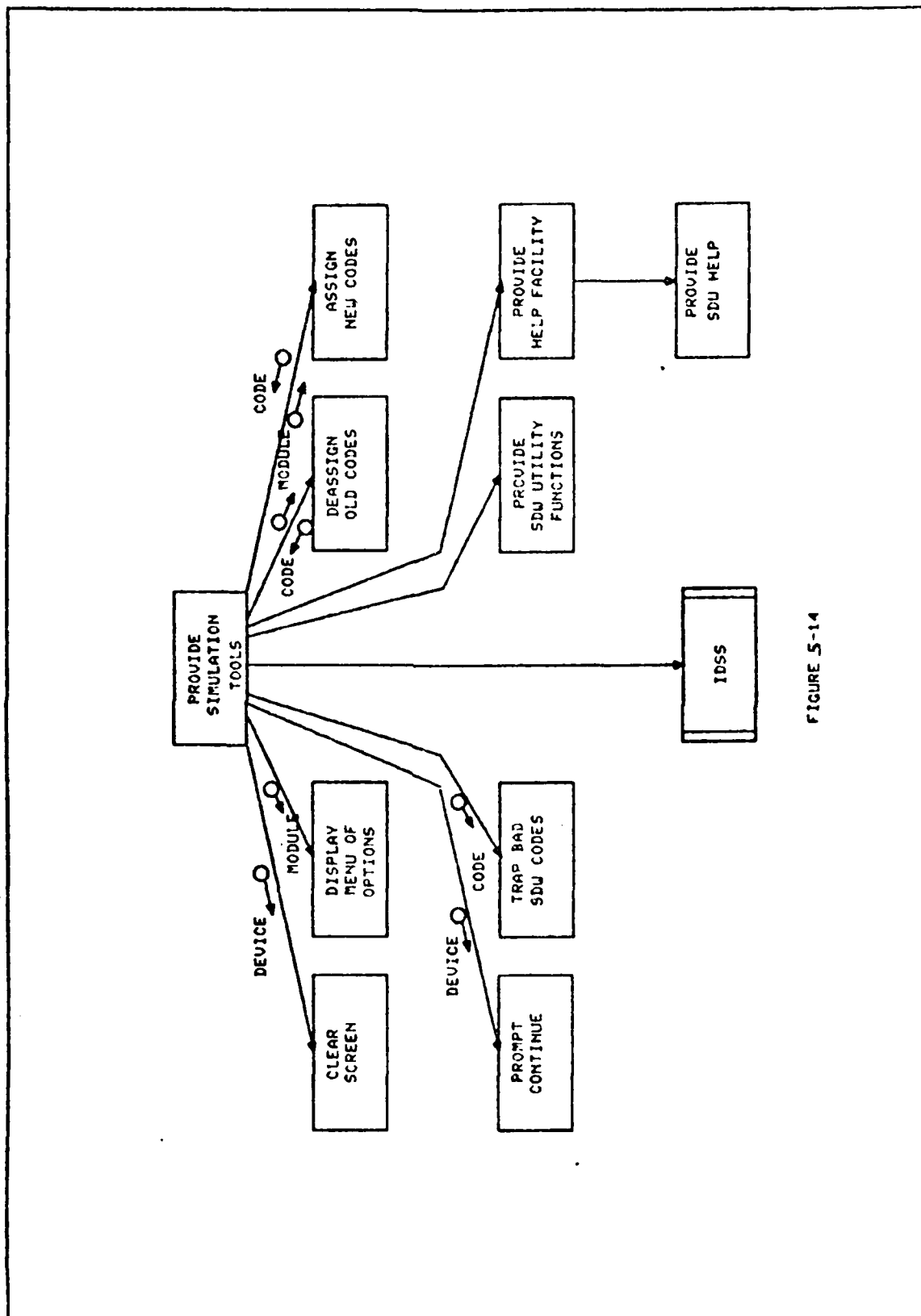


FIGURE 5-14

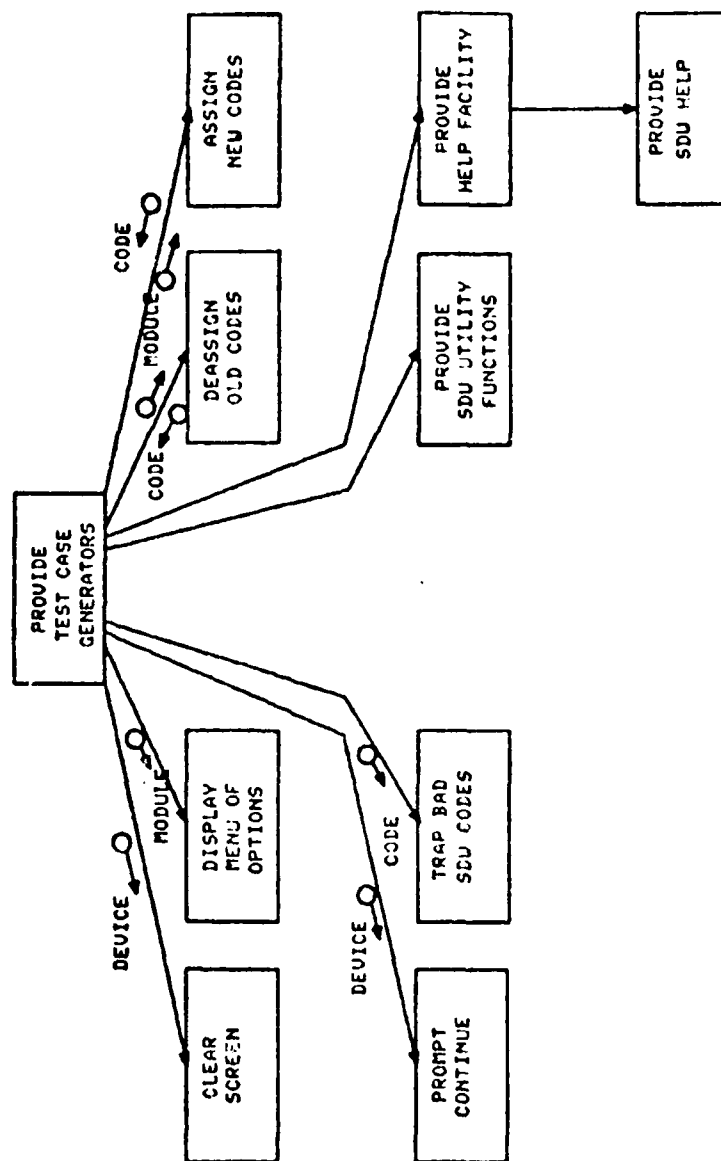


FIGURE 5-15

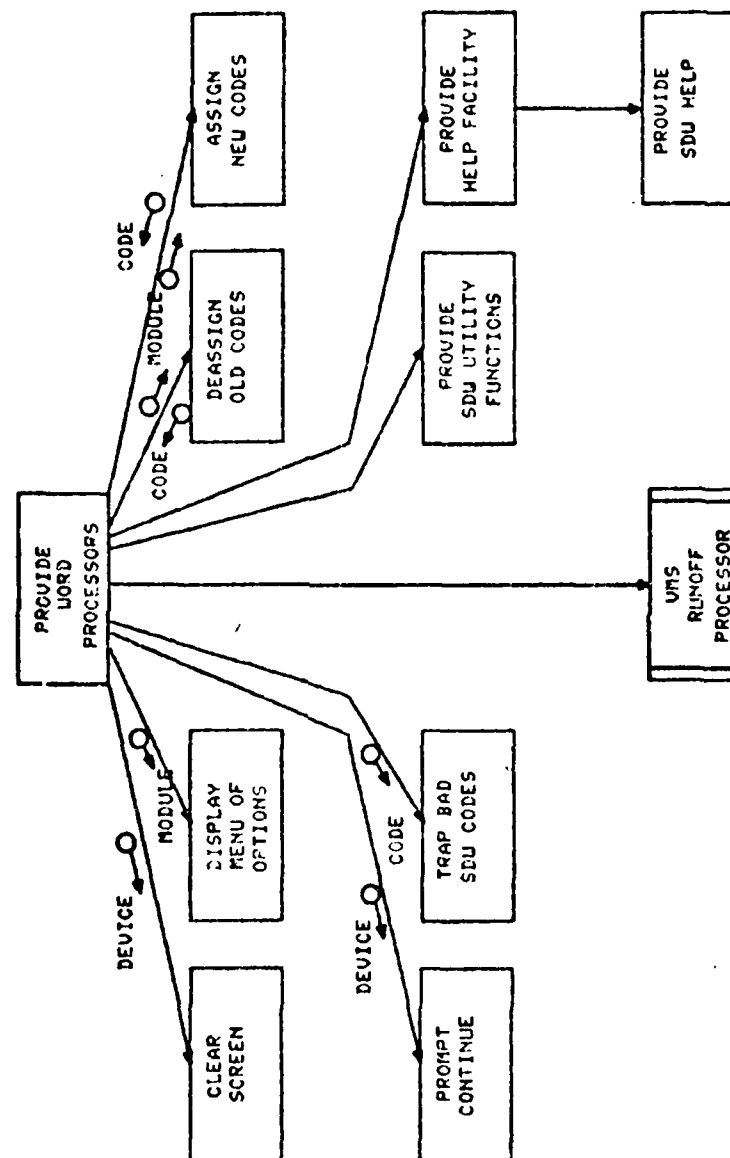


FIGURE 5-16

Appendix F:  
Algorithmic Design Of The  
Software Development Workbench Executive

### Algorithmic Design Of The Software Development Workbench Executive

Within the software life-cycle, the Detailed Design stage is characterized by the development of the design module algorithms. In order to better describe the objective of this stage, it is referred to as the Algorithmic Design stage in the rest of this document. The design modules for the SDWE are specified in the SDWE Preliminary Design model (Appendix E). Explicit algorithms for these design modules are developed using the SDWE Requirements model (Appendix D) for a reference.

The algorithms for the SDWE design modules are expressed in Structured English (Ref 90:48-49). The other Software Engineering Methodologies available for the specification of the algorithms are Decision Tables (Ref 90:49) and Decision Trees (Ref 90:49). Neither of these two options are very applicable to the development of the SDWE, whereas, the flexibility of Structured English makes it the most appropriate choice.

The subset of Structured English used to describe the SDWE algorithms uses a limited set of constructs, action verbs, data items, and other english words to formulate the algorithms in an easily understandable form. The Structured English constructs of the SDWE algorithms are the IF\_THEN, IF\_THEN\_ELSE, and REPEAT\_UNTIL. Structured English is a



very useful tool for describing algorithms because of its understandability. As a result, the reader who is unfamiliar with Structured English should be able to pick up it's concept without much difficulty.

The algorithms for the SDWE modules are displayed and explained in the text. Each of these algorithms are identified in the below.

#### SDWE Algorithms

Figure	Algorithm Title
6-1	SDWEXE
6-2	Functional Tool Group
6-3	List Project Data Bases
6-4	Access Pre-Fab Software Description Data Base
6-5	Help Facility
6-6	Trap Bad Commands
6-7	SDW Help Facilities
6-8	Provide SDW Utility Functions

The highest level design module of the SDWE Preliminary Design model is the SDWEXE module. This module must satisfy the requirement specifications defined by diagrams/activity boxes A0, A1, A2, A31, A35, A37 of the SDWE Requirements model (Appendix D). The algorithm for satisfying these requirements is detailed below:

### SDWEXE Algorithm

---

```
(* Initialize the SDW *)

WRITE SDW_Header_Message
GET Device_Type

(* Provide a set of Top Level Options *)

DEFINE Functional_Tool_Group_Codes
DEFINE Help_Command
DEFINE Menu_Command
DEFINE List_Project_DBs_Command
DEFINE Access_Pre-Fab_S/W_Descrip_DB
DEFINE Termination_Command

(* Accept and Execute the User Command *)

REPEAT_UNTIL User_Command equals Termination_Command
    IF Auto_Menu_Flag is true THEN
        DISPLAY Top_Level_Menu
    GET User_Command
    IF User_Command is invalid THEN
        CALL Trap_Bad_Commands
    EXECUTE User_Command
END_REPEAT_UNTIL
CLOSE Project_DB_Name
WRITE Conclusion_Message
```

Figure 6-1

Although each SDW Functional Tool Group controls and interfaces to a different set of SDW components, the manner in which each of the functional tool group modules perform their functions is very similar. As a result, a generic algorithm is provided for these modules. This algorithm is presented below:

### Functional Tool Group Module Algorithm

---

```
UNDEFINE Previous_Codes
DEFINE Functional_Tool_Group_Codes
DEFINE Help_Command
DEFINE SDW_Utility_Function_Command
DEFINE Menu_Command
DEFINE Return_Command
REPEAT_UNTIL User_Command equals Return_Command
    IF Auto_Menu_Flag is true THEN
        DISPLAY Current_Menu
    ELSE (Auto_Menu_Flag is false)
        SO DISPLAY Functional_Tool_Group_ID
    DEFINE Help_Request, Menu_Request, Return_Command
    GET User_Command
    IF User_Command is invalid THEN
        CALL Trap_Bad_Commands
    IF User_Command requires parameters THEN
        GET Parameters
    EXECUTE User_Command
END_REPEAT_UNTIL
UNDEFINE Functional_Tool_Group_Codes
RETURN to SDWEXE
```

Figure 6-2

There are several other design modules in the SDWE Preliminary Design model that must have algorithms specified for them. They are the List Project DBs module, the Access Pre-Fab S/W Descrip. DB module, the Help Facility Module, the SDW Help Facility module, and the Trap Bad Commands module. The algorithms for these modules are defined below:

### List Project DBs Algorithm

---

```
IDENTIFY Project_DB_Names
WRITE Header_Message
WHILE more Project_DB_Names
    WRITE next Project_DB_Name
END_WHILE
RETURN to SDWEXE
```

Figure 6-3

### Access the Pre-Fab S/W Descrip. DB Algorithm

---

```
UNDEFINE Previous_Codes
DEFINE Add_S/W_Descrip_Command
DEFINE Find_S/W_Descrip_Command
DEFINE Help_Command, Menu_Command, Return_Command
REPEAT_UNTIL User_Command equals Return_Command
    IF Auto_Menu_Flag is true THEN
        DISPLAY Current_Menu
    GET User_Command
    EXECUTE User_Command
END_REPEAT_UNTIL
UNDEFINE Add_S/W_Descrip_Command
UNDEFINE Find_S/W_Descrip_Command
RETURN to SDWEXE
```

Figure 6-4

### Help Facility Algorithm

---

```
GET Type_of_Help_Request
IF Type_of_Help_Request equals SDW_Help_Request THEN
    IF Auto_Menu_Flag is true THEN
        DISPLAY Menu of SDW_Help_Options
    GET SDW_Component_Selection
    IF SDW_Component_Selection is "SDW" THEN
        CALL SDW_Help_Facility
    ELSE
        DISPLAY Appropriate_Help_File
ELSE_IF Type_of_Help_Request equals
VMS_Help_Request THEN
    IF Auto_Menu_Flag is true THEN
        DISPLAY Menu of VMS_Help_Options
    GET VMS_Selection
    CALL VMS_Help_Facility
ELSE (* No help is really desired *)
    NO OPERATION
RETURN to calling module
```

Figure 6-5

### Trap Bad Commands Algorithm

---

```
DISPLAY Bad_Code
EXPLAIN Bad_Code
```

Figure 6-6

### SDW\_Help\_Facility

---

```
PROVIDE Menu_of_General_Help_Options
GET Help_Option
DISPLAY Requested_Help_File
```

Figure 6-7

PROVIDE\_SDW\_UTILITY\_FUNCTIONS

---

```
UNDEFINE Previous_Codes
DEFINE Utility_Functional_Codes
DEFINE Help_Command
DEFINE Menu_Command
DEFINE Return_Command
REPEAT_UNTIL User_Command equals Return_Command
  IF Auto_Menu_Flag is true THEN
    DISPLAY Current_Menu
  ELSE (Auto_Menu_Flag is false)
    SO DISPLAY Utility_Functions_ID
  GET User_Command
  IF User_Command is invalid THEN
    CALL Trap_Bad_Commands
  IF User_Command requires parameters THEN
    GET Parameters
  IF User_Command requires Device_Type change THEN
    DETERMINE new Device_Type
  IF User_Command requires
    Data_Storage_Scheme change THEN
    IF Project_DBs are required THEN
      GET Project_DB_Name
      IF Project_DB_Name does not
        already exist THEN
        CREATE Project_DB_Name
      SETUP Project_DB_Name
    ELSE IF directory desired for
      Data_Storage_Scheme THEN
      GET Directory_Spec
      SETUP Directory
    ELSE
      NO OPERATION
  IF User_Command requires Auto_Menu_Flag change THEN
    GET new Auto_Menu_Flag
END_REPEAT_UNTIL
UNDEFINE Utility_Function_Codes
RETURN to SDWEXE
```

Figure 6-8

APPENDIX G:  
Listing of the SDW Command Codes

### Listing of the SDW Command Codes

The following is a list of all of the legal SDW codes. This list is provided as a maintenance tool because the structure of the SDW and its associated text files require that each SDW command be given a unique code. Therefore, if new SDW commands are entered, this list will help insure that it will be given a unique code.

- AD - Run the AIDES Structured Design tool.
- AI - Run the Interim AUTOIDEF Drafting tool.
- AM - Alter the setting of the Auto\_Menu facility.
- AS - Add a software description to the Pre-Fab Software Description Data Base.
- BS - Run the BASIC compiler.
- CB - Run the COBOL compiler.
- CD - Continue an interrupted Debug session.
- CI - Run the CIDEF Design and Code Generation tool.
- CM - Enter the SDW Compiler functional group.
- CP - Enter the SDW Comparator functional group.
- DA - Enter the SDW Dynamic Analysis functional group.
- DB - Enter the SDW Debugger functional group.
- DE - Begin a Debug session
- DF - Run the VMS Differences comparator.
- DS - Enter the SDW Design Tool functional group.
- DV - Alter the User's device specification.
- ED - Enter the SDW Editor functional group.
- EE - Run the EDT editor.
- ER - Run the Extended Requirements Engineering and Validation System (EREVS).



- ES    - Run the SOS editor.
- EX    - Exit the current SDW command module.
- FR    - Run the FORTRAN compiler.
- FS    - Find a software description(s) in the  
Pre-Fab Software Description Data Base.
- GR    - Enter the SDW Graphics Editors functional  
group.
- HL    - Run the SDW Help Facility.
- IS    - Run the Integrated Decision Support System  
(IDSS).
- LI    - Run the VMS linker.
- LK    - Enter the SDW Linker functional group.
- LP    - List all Project Data Bases.
- MN    - Display the current menu of options.
- PD    - Specify a new data storage area.
- PF    - Enter the SDW Pre-Fab Software Description  
Data Base access module.
- PM    - Enter the SDW Performance Monitor functional  
group.
- PS    - Run the PASCAL compiler.
- RD    - Enter the SDW Requirements Definition functional  
group.
- RE    - Run the Requirements Engineering and Validation  
System (REVS).
- RN    - Run the RUNOFF text processor.
- SA    - Enter the SDW Static Analysis tools functional  
group.
- SD    - Run the AFIT Syntax-Directed Editor.
- SM    - Enter the SDW Simulation tool functional group.
- TC    - Enter the SDW Test Case Generator functional group.

UT - Access the SDW Utility Functions.

WP - Enter the SDW Word Processor functional group.

APPENDIX H:  
SDWE File Descriptions

## SDWE File Descriptions

This document is a listing and description of all of the SDW component files. It is to be used as a guide for future maintenance of the Software Development Workbench (SDW).

ADDSW.EXE;1 -The executable image of the application program used to add a software description into the Pre-Fabricated Software Description Data Base.

ADDSW.OBJ;1 -The object code of the application program used to add a software description into the Pre-Fabricated Software Description Data Base.

ADDSW.PAS;1 -The source code of the application program used to add a software description into the Pre-Fabricated Software Description Data Base.

ADHELP.DAT;1 -The RUNOFF input file for the help message for the AIDES Structured Design Tool.

ADHELP.MEM;1 -The help file for the AIDES Structured Design Tool.

AIHELP.MEM;1 -The help file for the Interim AUTOIDEF Drafting Tool.

AMHELP.DAT;1 -The RUNOFF input file for the Alter Auto\_menu facility help file.

AMHELP.MEM;1 -The help file for the SDW command used to alter the use of the auto\_menu prompting facility.

ASHELP.MEM;1 -The help file for the SDW command used to add a software description to the Pre-Fabricated Software Description Data Base.

ASSIGNSYM.COM;1 -The command module used to assign the SDW symbols for the current module.

BADCODE.COM;1 -The command module used to trap invalid inputs to the SDW.

BSHELP.MEM;1 -The help file for the SDW Basic Compiler.

BSPARMTS.DAT;1 -The file of optional qualifiers for the Basic Compiler.

CBHELP.MEM;1 -The help file for the SDW COBOL Compiler.

CBPAPMTS.DAT;1 -The file of optional qualifiers for the COBOL compiler.

CDHELP.DAT;1 -The RUNOFF input file for the help message dealing with continuing an interrupted Debug session.

CDHELP.MEM;1 -The help file for the command that continues and interrupted Debug session.

CIHELP.MEM;1 -The help file for the CIDEF Design and Code Generation facility.

CLEARSCRN.COM;1 -The command module that is used to clear the video display screen. (\*\* This module contains terminal dependent features \*\*)

CLEARTKTX.EXE;1 -The executable image of the module used to clear a Tektronix display screen.

CLEARTKTX.OBJ;1 -The object code of the module used to clear a Tektronix display screen.

CLEARTKTX.PAS;1 -The source code of the module used to clear a Tektronix display screen.

CMHELP.DAT;1 -The RUNOFF input file for the SDW compiler help file.

CMHELP.MEM;1 -The help file for the SDW compilers.

CMID.DAT;1 -The RUNOFF input file for the compiler ID message.

CMID.MEM;1 -The ID message for the SDW compiler functional group.

CMMENU.DAT;1 -The RUNOFF input file for the SDW compiler menu.

CMMENU.MEM;1 -The menu for the SLW compiler functional group.

COMFILE.DAT;1 -The list of all of the SDW command modules.

COMPARE.COM;1 -The command module for the SDW comparators.

COMPILE.COM;1 -The command module for the SDW compilers.

CONTINUE.COM;1 -The command module that prompts for and executes the user's command to continue with the SDW execution. (\*\* This module contains device dependent features \*\*)

CPHELP.DAT;1 -The RUNOFF input file for the SDW comparators help file.

CPHELP.MEM;1      -The SDW comparators help file.

CPID.DAT;1        -The RUNOFF input file for the SDW comparators functional group ID.

CPID.MEM;1        -The ID for the SDW comparator functional group.

CPMENU.DAT;1      -The RUNOFF input file for the SDW comparator's menu.

CPMENU.MEM;1      -The SDW comparator's menu.

DAHLP.DAT;1      -The RUNOFF input file for the SDW Dynamic Analysis Tools help message.

DAHLP.MEM;1      -The help file for the SDW Dynamic Analysis Tools functional group.

DAID.DAT;1        -The RUNOFF input file for the SDW Dynamic Analysis Tools functional group identification message.

DAID.MEM;1        -The SDW Dynamic Analysis Tool functional group ID.

DAMENU.DAT;1      -The RUNOFF input file for the SDW Dynamic Analysis Tools menu.

DAMENU.MEM;1      -The menu for the SDW Dynamic Analysis Tools functional group.

DATOLS.COM;1      -The command module for the SDW Dynamic Analysis Tools functional group.

DBHELP.DAT;1      -The RUNOFF input file for the SDW Debuggers help message.

DBHELP.MEM;1      -The help file for the SDW Debuggers functional group.

DBID.DAT;1        -The RUNOFF input file for the SDW Debuggers ID message.

DBID.MEM;1        -The SDW Debugger functional group ID message.

DBMENU.DAT;1      -The RUNOFF input file for the SDW Debuggers menu.

DBMENU.MEM;1      -The menu file for the SDW Debugger functional group.

DEBUGGER.COM;1    -The command module for the SDW Debugger functional group.

DEHELP.DAT;1      -The RUNOFF input file for the Begin Debug help file.

DEHELP.MEM;1      -The Begin Debug command help file.

DELSYMBOL.COM;1 -The command module that deletes the SDW symbols for  
modules that are about to be exited.

DFHELP.MEM;1      -The help file for the VMS Differences Facility.

DFPARMTS.DAT;1 -The file of optional qualifiers for the VMS  
Differences Facility.

DSGNTOL.COM;1 -The command module for the SDW Design Tools  
functional group.

DSHELP.DAT;1      -The RUNOFF input file for the SDW Design Tools help  
message.

DSHELP.MEM;1      -The SDW Design Tools functional group help file.

DSID.DAT;1         -The RUNOFF input file for the SDW Design Tools ID.

DSID.MEM;1         -The ID for the SDW Design Tool functional group.

DSMENU.DAT;1      -The RUNOFF input file for the SDW Design Tool menu.

DSMENU.MEM;1      -The menu file for the SDW Design Tool functional  
group.

DSPLMENU.COM;1 -The command module that displays on the user's  
terminal the current menu of SDW options.

DVHELP.DAT;1         -The RUNOFF input file for the help file for the  
altering user's device spec module.

DVHELP.MEM;1 -The help file for the altering user's device spec  
module.

EDHELP.DAT;1      -The RUNOFF input file for the SDW Editors help  
message.

EDHELP.MEM;1      -The help file for the SDW Editors functional group.

EDID.DAT;1         -The RUNOFF input file for the SDW Editor's ID  
message.

EDID.MEM;1         -The SDW Editors functional group ID message.

EDITALL.COM;1      -The command module used during SDW development to  
edit a group of similar files.

EDITORS.COM;1      -The command module for the SDW Editors functional  
group.

EDMENU.DAT;1 -The RUNOFF input file for the SDW Editors menu.

EDMENU.MEM;1 -The menu file for the SDW Editors functional group.

ELHELP.MEM;1 -The help file for the SOS editor.

ELPARMTS.DAT;1 -The file holding the optional qualifiers for the SOS editor.

ERHELP.MEM;1 -The help file for the Extended Requirements Engineering and Validation System (EREVS).

ESHELP.MEM;1 -The help file for the EDT editor.

ESPARMTS.DAT;1 -The file holding the optional qualifiers for the EDT editor.

EXHELP.DAT;1 -The RUNOFF input file for the SDW "EX" command help message.

EXHELP.MEM;1 -The help file for the SDW "EX" command.

FILES.DAT;1 -The RUNOFF input file for this listing of files.

FILES.MEM;1 -This file of SDW file names and descriptions.

FINDSW.EXE;1 -The executable image of the application program used to find a software description in the Pre-Fabricated Software Description Data Base.

FINDSW.OBJ;1 -The object code for the application program used to find a software description in the Pre-Fabricated Software Description Data Base.

FINDSW.PAS;1 -The source code for the application program used to find a software description in the Pre-Fabricated Software Description Data Base.

FRHELP.MEM;1 -The help file for the SDW FORTRAN Compiler.

FRPARMTS.DAT;1 -The file holding the optional qualifiers for the FORTRAN compiler.

FSHELP.MEM;1 -The help file for the SDW command used to find a software description in the Pre-Fabricated Software Description Data Base.

GRAPHICS.COM;1 -The command module for the SDW Graphics Editors functional group.

GRHELP.DAT;1 -The RUNOFF input file for the SDW Graphics Editors help file.



GRHELP.MEM;1      -The help file for the SDW Graphics Editors functional group.

GRID.DAT;1        -The RUNOFF input file for the SDW Graphics Editors functional group ID.

GRID.MEM;1        -The ID file for the SDW Graphics Editors functional group.

GRMENU.DAT;1      -The RUNOFF input file for the SDW Graphics Editors functional group menu.

GRMENU.MEM;1      -The menu file for the SDW Graphics Editors function group.

HEADER.DAT;1      -The RUNOFF input file for the header for the SDWE documentation package.

HEADER.MEM;1      -The header for the SDWE documentation package.

HELPER.COM;1      -The command module that implements the SDW Help Facility.

HELPPFILE.DAT;1   -The directory listing of all help files.

HLHELP.DAT;1      -The RUNOFF input file for the help message on the SDW Help Facility.

HLHELP.MEM;1      -The help file for the SDW Help Facility.

IDFILE.DAT;1      -The directory listing of all ID files.

INSTALL.DAT;1     -The RUNOFF input file that contains the SDWE installation procedures.

INSTALL.MEM;1     -The file that contains the installation procedures for the SDWE.

ISHelp.MEM;1      -The help file for the Integrated Decision Support System (IDSS).

LIHELP.MEM;1      -The help file for the SDW command to use the linker.

LINKER.COM;1      -The command module for the SDW Linkers.

LIPARMTS.DAT;1    -The file holding the optional qualifiers for the VMS linker.

LISTPDBS.COM;1    -The command module that reports a list of all existing Project Data Bases.

LKHELP.MEM;1        -The help file for the SDW Linkers functional group.

LKID.DAT;1         -The RUNOFF input file for the SDW Linkers ID message.

LKID.MEM;1               -The ID message for the SDW Linkers functional group.

LKMENU.DAT;1        -The RUNOFF input file for the SDW Linkers menu.

LKMENU.MEM;1        -The menu file for the SDW Linkers functional group.

LPHELP.DAT;1        -The RUNOFF input help file for the SDW command to report all existing Project Data Bases.

LPHELP.MEM;1        -The help file for the "LP" command, to report all existing Project Data Bases.

MAINT.DAT;1        -The RUNOFF input file that contains maintenance tips on the SDW.

MAINT.MEM;1        -The file that contains instructions and tips on potential maintenance situations for the SDW.

MENUFIL.DAT;1       -The directory listing of all menu files.

MNHELP.DAT;1        -The RUNOFF input file for the help file on SDW menus.

MNHELP.MEM;1        -The help file on SDW menus.

PDHELP.DAT;1       -The RUNOFF input file for the help file for the specifying of data storage areas command module.

PDHELP.MEM;1       -The help file for the specifying of data storage areas command module.

PERFMON.COM;1       -The command module for the SDW Performance Monitors.

PFDB.COM;1          -The command module for accessing the Pre-Fabricated Software Description Data Base.

PFHELP.DAT;1        -The RUNOFF input file for the SDW Pre-Fab Software Description Data Base help file.

PFHELP.MEM;1        -The help file for the SDW Pre-Fab Software Description DB.

PFID.DAT;1          -The RUNOFF input file for the SDW Pre-Fab

Software Description DB ID message.

PFID.MEM;1        -The ID message for the SDW Pre-Fab Software Description DB.

PFMENU.DAT;1      -The RUNOFF input file for the SDW Pre-Fab Software Description Data Base menu of options.

PFMENU.MEM;1      -The menu file for the SDW Pre-Fab Software Description DB.

PMHELP.DAT;1      -The RUNOFF input file for the SDW Performance Monitors help message.

PMHELP.MEM;1      -The help file for the SDW Performance Monitor's functional group.

PMID.DAT;1        -The RUNOFF input file for the SDW Performance Monitors ID.

PMID.MEM;1        -The ID message for the SDW Performance Monitors functional group.

PMMENU.DAT;1      -The RUNOFF input file for the SDW Performance Monitors menu.

PMMENU.MEM;1      -The menu file for the SDW Performance Monitors functional group menu.

PSHELP.MEM;1      -The help file for the SDW PASCAL Compiler.

PSPARMTS.DAT;1    -The file of optional qualifiers for the PASCAL Compiler.

RDHELP.DAT;1      -The RUNOFF input file for the SDW Requirements Definition help file.

RDHELP.MEM;1      -The help file for the SDW Requirements Definition functional group.

RDID.DAT;1        -The RUNOFF input file for the SDW Requirements Definition ID.

RDID.MEM;1        -The ID message for the SDW Requirements Definition functional group.

RDMENU.DAT;1      -The RUNOFF input file for the SDW Requirements Definition tools menu.

RDMENU.MEM;1      -The menu file for the SDW Requirements Definition Tools.

REHELP.MEM;1      -The help file for the Requirements Engineering and Validation System (REVS).

REQDEF.COM;1      -The command module for the SDW Requirements Definition Tools functional group.

RNHELP.MEM;1      -The help file for the RUNOFF text processor facility.

RNPARMTS.DAT;1    -The file of optional qualifiers for the RUNOFF facility.

SAHELP.DAT;1      -The RUNOFF input file for the SDW Static Analysis Tools help file.

SAHELP.MEM;1      -The help file for the SDW Static Analysis Tools.

SAID.DAT;1        -The RUNOFF input file for the SDW Static Analysis Tools ID.

SAID.MEM;1        -The ID message for the SDW Static Analysis Tools.

SAMENU.DAT;1      -The RUNOFF input file for the SDW Static Analysis Tools menu.

SAMENU.MEM;1      -The menu file for the SDW Static Analysis Tools.

SATOOLS.COM;1     -The command module that controls the SDW Static Analysis Tools.

SDHELP.DAT;1      -The RUNOFF input file for the help file on the SDW Syntax-Directed Editor.

SDHELP.MEM;1      -The help file on the SDW Syntax-Directed Editor.

SDWCODES.DAT;1    -The RUNOFF input file for a listing of all SDW commands.

SDWCODES.MEM;1    -The listing of all SDW commands.

SDWEXE.COM;1      -The top-level SDW command module.

SDWHELP.COM;1     -The command module that provides general help on the SDW.

SDWHELP.DAT;1     -The RUNOFF input file for the GENERAL SDW help file.

SDWHELP.MEM;1     -The help file for GENERAL SDW help.

SDWHELP2.DAT;1    -The RUNOFF input file for the STRUCTURE SDW help file.

SDWHELP2.MEM;1 -The help file for STRUCTURE SDW help.

SDWHELP3.DAT;1 -The RUNOFF input file for the 1st OPERATION SDW help file.

SDWHELP3.MEM;1 -The help file for the 1st OPERATION SDW help.

SDWHELP4.DAT;1 -The RUNOFF input file for the 2nd OPERATION SDW help file.

SDWHELP4.MEM;1 -The help file for the 2nd OPERATION SDW help.

SDWHELP5.DAT;1 -The RUNOFF input file for the 3rd OPERATION SDW help file.

SDWHELP5.MEM;1 -The help file for the 3rd OPERATION SDW help.

SDWHLMN.DAT;1 -The RUNOFF input file for the menu of SDW Help options.

SDWHLMN.MEM;1 -The menu file for high level SDW help options.

SDWMENU.DAT;1 -The RUNOFF input file for the SDW Top-Level menu.

SDWMENU.MEM;1 -The menu file for the SDW Top-Level menu.

SDWUTIL.COM;1 -The command module that controls access to the SDW Utility Functions.

SETAIDEF.COM;1 -The command module that controls the Interim AUTOIDEF tool.

SETAIDES.COM;1 -The command module that controls the AIDES Structured Design tool.

SETCIDEF.COM;1 -The command module that controls the CIDEF Design and Code Generation tool.

SETEREVS.COM;1 -The command module that controls the EREVS Requirements tool.

SETIDSS.COM;1 -The command module that controls the IDSS simulation tool.

SETREVS.COM;1 -The command module that controls the REVS Requirements tool.

SETSDW.COM;1 -The command module that set up the required logical names to run the SDW.

SFHELP.DAT;1 -The RUNOFF input file for the SYSFLOW tool

help message.

SFHELP.MEM;1 -The SYSFLOW tool help message file.

SIMULATE.COM;1 -The command module that controls the SDW Simulation tools.

SMHELP.DAT;1 -The RUNOFF input file for SDW Simulation tool's help message.

SMHELP.MEM;1 -The help file for the SDW Simulation tools.

SMID.DAT;1 -The RUNOFF input file for the SDW Simulation tools ID.

SMID.MEM;1 -The ID message for the SDW Simulation tools.

SMMENU.DAT;1 -The RUNOFF input file for the SDW Simulation tools menu.

SMMENU.MEM;1 -The menu file for the SDW Simulation tools.

TCHELP.DAT;1 -The RUNOFF input file for the SDW Test Case Generators help message.

TCHELP.MEM;1 -The help file for the SDW Test Case Generators.

TCID.DAT;1 -The RUNOFF input file for the SDW Test Case Generator's ID.

TCID.MEM;1 -The ID message for the SDW Test Case Generator tools.

TCMENU.DAT;1 -The RUNOFF input file for the SDW Test Case Generator menu.

TCMENU.MEM;1 -The menu file for the SDW Test Case Generators.

TSTCASGEN.COM;1 -The command module that controls the SDW Test Case Generator tools.

USERMAN.DAT;1 -The RUNOFF input file for the SDWE User's Manual

USERMAN.MEM;1 -The SDWE User's Manual.

UTHELP.DAT;1 -The RUNOFF input file for the UT module help file.

UTHELP.MEM;1 -The help file for the SDW Utility Functions.

UTID.DAT;1 -The RUNOFF input file for the ID message for UT.

UTID.MEM;1 -The SDW Utility Functions ID message.

UTMENU.DAT;1 -The RUNOFF input file for the UT menu.

UTMENU.MEM;1 -The menu file for the SDW Utility Functions.

WELCOME.DAT;1 -The RUNOFF input file for the initial SDW header.

WELCOME.MEM;1 -The initial SDW header message.

WORDPROC.COM;1 -The command module that controls the SDW Word Processors.

WPHELP.DAT;1 -The RUNOFF input file for the SDW Word Processor's help message.

WPHELP.MEM;1 -The help file for the SDW Word Processors.

WPID.DAT;1 -The RUNOFF input file for the SDW Word Processor's ID.

WPID.MEM;1 -The ID message for the SDW Word Processors.

WPMENU.DAT;1 -The RUNOFF input file for the SDW Word Processor menu.

WPMENU.MEM;1 -The menu file for the SDW Word Processors.

Appendix I:

SDWE User's Manual



SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE (SDWE)  
USER'S MANUAL

(C) Copyright 1982 by  
Lt. Steven Hadfield  
Dr. Gary B. Lamont

SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE (SDWE)

TABLE OF CONTENTS  
-----

<u>TITLE</u>	<u>PAGE</u>
1. GENERAL INFORMATION ON THE SDWE	339
2. A WALK-THROUGH OF THE SDWE IN OPERATION	341
2.1 SDW INITIAL PROCEDURES	341
2.2 SDW TOP LEVEL OPERATION	342
2.3 SDW FUNCTIONAL TOOL GROUPS OPERATION	344
2.4 SDW HELP FACILITY	345
3. CONCLUDING INFORMATION	346

## SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE (SDWE) USER'S MANUAL

The Software Development Workbench Executive (SDWE) is a top-down, menu-driven interface to the Software Development Environment (SDW). The SDW is a software engineering environment that has many automated and interactive tools to support the development and maintenance of software. However, many of the SDW components are equally useful for the development of systems in general.

The objective of this user's manual is to instruct new users on the operation of the SDWE. This objective is achieved by presenting a general description of the SDWE and then walking the new user through the operation of the SDWE.

### 1. GENERAL INFORMATION ON THE SDWE

As previously mentioned, the SDWE is a top-down, menu-driven interface to the SDW. The SDWE is designed to run on any VAX compatible terminal. The highest level of the SDWE is the SDW Top-Level. After a short initiation section, the user is presented with the SDW Top-Level menu. From this level the SDW user may choose any of fourteen (14) functional tool groups or one of a number of other SDW commands. In addition, the user may use any DEC Command Language (DCL) command, provided the entire command is

entered on a single line.

If a functional tool groups is selected, the user is presented with another menu of options that are specific to that functional tool group. All of the SDW component tools are run from this functional tool group level. Any of the DCL commands may also be run from this level, (however, they must still be entered on a single line).

At the SDW Top-Level and at all of the functional tool groups, only the SDW menu options presented at that level and the DCL commands are executable. There are a few commands that are recognized at both the SDW Top-Level and the functional tool groups. They are the "HL" command that provides the user with automated help on the SDW; the "UT" command that allows the user to change the auto-menu prompting facility, select a Project Data Base for data storage, or change the terminal device specification for his terminal; the "MN" command that displays the current menu of options; and the "EX" command that causes an exit from the current SDW level.

## 2. A WALK-THROUGH OF THE SDWE IN OPERATION

This walk-through of the SDW in operation is broken into four sections. The first section deals with the initial set up procedures for the SDW. The second section illustrates the options available at the SDW Top-Level. The third section illustrates the options available within the SDW functional tool groups. The fourth and final section deals with the SDW Help Facility.

### 2.1 SDW INITIAL PROCEDURES

The Software Development Workbench (SDW) is entered with the "SDW" command given from the monitor level of the operating system. The SDW then presents the user with a header message that indicates the version number of the SDW.

The SDW user is then presented with a query as follows:

Enter the type of terminal in use (VT52,VT100,4014,4016)

If your device is not listed above hit a <RET> :

The SDW requires a specification of the type of terminal device that the SDW user is working from. The SDW recognizes the DEC VT52, the DEC VT100, the Tektronix 4014, and the Tektronix 4016. If your device is not listed, or you are unsure of the type of device you are using, simply hit a <RET> (carriage return). The default for the device

type is a DEC VT52. After the completion of this initial procedure, the user is at the SDW Top-Level.

## 2.2 SDW TOP LEVEL OPERATION

The SDW Top Level is indicated by the menu title and/or by the special command prompt "SDW> :". At this prompt, any of the options indicated on the menu may be selected, or any DCL command may be entered on a single line. There are twenty SDW options at the SDW Top Level. Fourteen of these options call up SDW functional tool groups (whose operation is detailed in the next section of this manual). The other six command options provide means to 1) access the Pre-Fabricated Software Description Data Base (that holds the descriptions and locations of existing software modules), 2) obtain a listing of all of the existing Project Data Bases, 3) access the SDW Help Facility (for help on any SDW command or any DCL command), 4) perform one of the SDW Utility Functions, 5) display the SDW Top Level menu, and 6) exit the SDW and return to the operating system monitor level. All of the SDW options are selected by entering the two letter code for that option after the "SDW> :" prompt.

The "UT" command at this level allows the SDW user to access any of three very important SDW Utility Functions. The first of these functions is the "AM" function that

provides the user with the option of being automatically prompted with the current set of options at each level. The SDW will prompt the SDW user with menus unless this function is executed. This function also provides for the resetting of the automatic menu facility if it was previously turned off.

The second of the SDW Utility Functions is the "DV" command. After the SDW user initially reports the type of terminal he is using to the SDW, he may wish to reset this specification. He does this with the "DV" command that re-prompts the SDW user for the device type. Again, the SDW recognizes only four types of terminals (VT100, VT52, 4014, 4016). If the SDW user is using a different type of terminal, he need only hit a <RET> at the prompt for a device type.

The final SDW Utility Function is the "PD" command. This function allows the SDW user to specify a data storage area for the products of the SDW. The SDW user may choose to use a Project Data Base (which is simply a separate directory under the SDW's main directory), or the user's default directory at the time he called up the SDW. With this function, the SDW user may change his data storage area to any Project Data Base or his default directory at any time. If a Project Data Base is selected, the name for it must be one to eight alphabetic characters.

### 2.3 SDW FUNCTIONAL TOOL GROUPS OPERATIONS

Within each of the functional tool groups, entered from the SDW Top Level, the user is presented with a new set of SDW command options. Only these options and the DCL commands are recognizable from these levels. The command prompts at each of the functional tool groups consist of the two letter code for that functional tool group followed by a ">:" in the following format "xx>:". The options within these functional tool groups are selected by entering the two letter code for that option after the command prompt. There are four command options common to all of the SDW functional tool groups. They are 1) "UT" for accessing the SDW Utility Functions, 2) "HL" for accessing the SDW Help Facility, 3) "MN" for displaying the current menu of options, and 4) "EX" to return to the SDW Top Level. The fourteen functional tool groups within the SDW are Compilers, Comparators, Dynamic Analysis Tools, Debuggers, Design Tools, Editors, Graphics Editors, Linkers, Performance Monitors, Requirements Definition Tools, Static Analysis Tools, Simulation Tools, Test Case Generators, and Word Processors. Information on the exact contents of each of these functional groups is available through the SDW Help Facility.



## 2.4 SDW HELP FACILITY

The SDW Help Facility is an on-line capability for providing general or selective help on the SDW and its components. The SDW Help Facility is accessed by entering the "HL" command from the SDW Top Level or any of the SDW functional tool group levels. Upon entering the SDW Help Facility, the user is prompted with the query:

Do you wish help on an SDW Component (Y or N):

SDW Components are defined as any of the two letter command options presented on any of the SDW menus. If the "Y" response is chosen and the Automatic Menu Prompting facility is enabled, the user will receive a menu of current help options. With or without the Automatic Menu Prompting facility, the user will be prompted with a request for the two letter code he is interested in:

Enter the 2-Letter SDW Component Code

(or enter 'SDW' for more general information):

The user is then provided with a help display on his selected option. The response "SDW" is also a legal option for this query and will provide the user with a selection of more general type information on the SDW. If the "SDW" response is entered, the user will be prompted with four high level help options. They are "BA" for background information on the SDW, "ST" for information on the

structure of the SDW, "OP" for information on the operation of the SDW, and "AL" for the displaying of all three of the help texts. At the end of each text message, the user is prompted to type a carriage return to continue.

The other response option to the initial Help Facility query is "N". This response will cause the following query to be displayed.

Do you wish help on a VMS facility (Y or N):

A "Y" response to this query puts the user into the VMS help facility. This facility gives the user a list of topics on which he may receive further help.

From within the SDW, a wide range of help facilities are accessible and provide complete on-line help capabilities for the user that wishes to learn by doing.

### 3. CONCLUDING INFORMATION

The SDWE was developed as a user-friendly interface to a variety of automated and interactive tools for the development of software systems and systems in general. This user's manual is provided as off-line assistance on the operation of the SDWE.

APPENDIX J:

SDWE Installation Guide

SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE  
INSTALLATION GUIDE

(C) Copyright 1982 by  
Lt. Steven Hadfield  
Dr. Gary B. Lamont

SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE (SDWE)  
INSTALLATION GUIDE

The Software Development Workbench Executive (SDWE) is a top-down, menu-driven user interface to the Software Development Workbench's component tools. The purpose of this document is to provide detailed instruction on the installation of the SDWE on the VAX-11/780 computer. These installation procedures assume the target VAX-11/780 is running under the VMS operating system. The SDWE is stored on a VMS formatted magnetic tape.

INSTALLATION INSTRUCTIONS

1. The first step in the installation of the SDWE is to create a top level directory entitled [SDW]. This can be done when logged in under "SYSTEM". The command for this operation is:

\$CREATE/DIR [SDW]

There are various options to this command which may be important to a specific installation. These options are documented in the VMS Help Facility and may be realized by the command:

\$HELP CREATE/DIR

IMPORTANT: The Project Data Bases of the SDW are data storage areas which may optionally be used by users to hold their development data. The Project Data Bases are currently implemented as sub-directories under the main directory [SDW]. Since the SDW users must have full read, write, edit, and delete privileges on these sub-directories, the [SDW] directory must be created with these privileges given at either the Group or World levels.

2. The next step in the installation process is to copy the SDWE code from the magnetic tape. The device name for a magnetic tape drive will be "MSxx:" where xx is the channel and device number. The "SHOW DEVICE M" command will display the tape drives on your system. Pick one not in use and use that device name where you see "MSxx" in these instructions.

The following sequence of DEC Command Language (DCL) commands is used to copy the SDWE from the magnetic tape.

```
$SET DEF disk_name:[SDW]    (* SETS THE PROPER DEFAULT
                             DISK DEVICE AND DIRECTORY *)
```

```
$ALLOCATE MSxx:             (* RESERVES THE TAPE DRIVE *)
```

-HAVE THE TAPE MOUNTED ON THE TAPE DRIVE, THE OPERATOR  
SHOULD ASSIST YOU WITH THIS.

```
$MOUNT MSxx: SDWE1 SDWE1
```

\$COPY/LOG MSxx:[SDW]\*.\* [SDW]\*.\*

\$DISMOUNT MSxx: (\* FINISHED WITH THE TAPE \*)

\$DEALLOCATE MSxx:

At this point all of the SDWE files are in the directory [SDW]. A description of each file is found in the file "FILES.DAT". Instructions on the SDWE's operation are found in file "USERMAN.MEM". A maintenance guide is in file "MAINT.MEM" and this document describing the installation procedures is found in file "INSTALL.MEM".

3. The next step in the installation procedure is to set up the logical names and symbols that are required to execute the SDWE and the SDW components. The device location for the SDWE and each of the SDW components are assigned logical names in the file "SETSDW.COM". You will need to edit this file and insure that each of the logical assignments is made to the proper device.

4. The final step in the installation of the SDWE is to include the following command into each of the SDW users' LOGIN.COM files.

\$@disk:[SDW]SETSDW

Where "disk" the resident device name for the SDWE. The use of this command insures that the proper symbols and logical names are defined for each SDW user.

APPENDIX K:

SDWE Maintenance Guide



SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE  
MAINTENANCE GUIDE

(C) Copyright 1982 by  
Lt. Steven Hadfield  
Dr. Gary B. Lamont

SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE  
MAINTENANCE GUIDE

TABLE OF CONTENTS  
-----

SECTION -----	PAGE -----
1. INTRODUCTION	355
1.1 GENERAL SDWE STRUCTURE	355
1.2 ON-LINE DOCUMENTATION STANDARD	356
2. SDWE MAINTENANCE AREAS	357
2.1 ADDITION OF NEW SDW COMPONENTS	358
2.2 ADDITION OF NEW VIDEO TERMINALS	359
2.3 PROJECT DATA BASE POTENTIAL PROBLEMS AND SOLUTIONS	361
2.4 ALTERING A TOOL'S RESIDENT DEVICE	362
2.5 LIMITED DEPTH OF SDWE CALLING	363
2.6 SPECIAL LIMITATIONS TO THE SDWE	363
3. CONCLUSION	364

# SOFTWARE DEVELOPMENT WORKBENCH EXECUTIVE MAINTENANCE GUIDE

## 1. INTRODUCTION

The Software Development Workbench Executive (SDWE) is a top-down, menu-driven interface to the Software Development Workbench (SDW). The SDW is a collection of automated and interactive tools that support the life-cycle development activities of system (software systems in particular). The SDWE is the interface and controller of the SDW. The SDWE is primarily written in the DEC Command Language (DCL), however, there are some low level modules written in PASCAL. The SDWE has been designed to run on any VAX 11/780 compatible video terminal. There are, however, special facilities to be realized when using VT100 terminals, or Tektronix 4014 s or 4016 s. The SDWE is also easily modifiable for special features when using other types of video terminals (see Section 2.2).

### 1.1 SDWE GENERAL STRUCTURE

As previously mentioned, the SDWE has a top-down structure. At the top of this structure is the SDW Top Level (SDWEXE.COM;1). This module provides the initial and

high level interface for the SDW. The SDW Top Level module sets up the execution of the SDW and provides for the execution of a variety of SDW commands. The set up of the SDW includes the establishing up of a Project Data Base if one is to be used. From the SDW Top Level module, commands can be selected to enter a SDW functional group. These groups provide access to the SDW component tools and form the second level of the SDW structure. The SDW functional groups may execute the tools directly or they may call other command modules that provide detailed set up for and execution of the SDW tools. Besides these modules, the SDWE has a few more modules that provide primitive functions, such as clearing the video display, prompting for a continue, and displaying a menu of options.

## 1.2 SDWE ON-LINE DOCUMENTATION STANDARDS

The bulk of the detailed documentation on the SDWE resides with in the coded modules. A specific standard was used for this documentation. The standard calls for the following header information on each module.

MODULE NAME

LAST DATE OF MODIFICATION

AUTHOR

CALLING MODULES

CALLED MODULES

FUNCTIONAL DESCRIPTION OF THE MODULE

Besides this header information, there are comment statements for each code section with the body of the module. This allows those unfamiliar with DCL to understand the code.

## 2. SDWE MAINTENANCE AREAS

Version 1.0 of the Software Development Workbench (SDW) is the initial prototype of an environment designed for the Air Force Institute of Technology. As a result of its developing nature, there are a few sections of the SDW that will quite likely require maintenance and modification. The following sections describe some anticipated modifications to specific sections of the SDW. They are provided to aid installation personnel in maintaining the SDW at their location.

## 2.1 ADDITION OF NEW SDW COMPONENTS

There are presently only a few operation components incorporated into the Software Development Workbench. Many additional tools are anticipated to be added into the SDW structure and for this reason a set of instruction on how to add a new tool are provided below.

1- Select a new 2-letter code that will be used to call up the new tool. This code must be unique. (A set of the present codes is included as Part 5 of the SDW Documentation Package.)

2- Create a logical name for the device that the new tool will reside on. An example would be "IDEF\$DISK:". Then, add a new assignment statement into the file "SETSDW.COM". This file is used to set up all of the SDW logical names for each one of the SDW users.

3- If the new tool requires special set up commands, a special command file must be written to facilitate this. An example of such a command file would be "SETEREVS.COM;1".

4- Then, the module for the functional group to which the new tool will belong must be edited

if the tool requires any special qualifiers or file specs. The "ASSIGNSYM.COM" must also be updated with the global definition of the two-letter code for the tool. This is done under the section of codes for that tool group. The "DELSYMBOL.COM" file is similarly update with a global delete command under the section reserved for the proper tool group.

5- The menu for the functional group must then be updated. The menu file will be named with the two letter code for the functional group followed by the letters "MENU.MEM".

6- The final step is to create a help file for the new tool. This should be done in the format of the other help files. The naming convention for this help file is the 2-letter code for the tool followed by "HELP.MEM". This file must be in the [SDW] directory.

## 2.2 ADDITION OF NEW VIDEO TERMINALS

Since many of the SDW component tools require special video terminal in order to function, the SDWE has been designed to be compatible to any terminal that can be otherwise used with the VAX 11/780. Since many of these

terminals have extensive display capabilities, the SDWE is easily modified to exploit device-dependent features. Presently, the SDWE uses a clear screen command extensively with VT100 and Tektronix 4014 and 4016 terminals. The SDWE also uses a reverse-video display for prompting for continuing on the VT100 terminals. These device-dependent features are buried within SDWE primitive modules. The type of the current user's device is stored in a SDWE global variable called "DEVICE". The SDW primitives will conditionally executed device-dependent features depending on the status of the "DEVICE" variable. The "DEVICE" variable is set in the SDW Top Level module located in file "SDWEXE.COM". Originally, the type of the device was determined automatically by the SDW Top Level module using a "SHOW TERMINAL" command. However, the device name shown by this command is not always the actual device being used. As a result, it was necessary to use queries to the user to determine the type of his device.

If you desire to modify the SDWE to provide special features for a new type of video terminal, you may follow the instructions provided below.

- 1- Add the terminal type to the device queries in "SDWEXE.COM" and "SDWUTIL.COM".

- 2- Update the SDWE primitive module(s) that deals



with the special feature(s) you wish to add.  
If no primitive module exists for that feature,  
create a new primitive module. Then, insert calls  
to that module into the SDWE command modules as  
appropriate. The new primitives must provide  
means to handle any type of device.

### 2.3 PROJECT DATA BASE POTENTIAL PROBLEMS AND A SOLUTION

The present implementation of the Project Data Bases is very elementary, they are simply sub-directories under [SDW]. Potential problems with this structure are that two different users may be unknowingly be using the same Project Data Base or the need for Read, Write, Edit, and Delete protection on the SDWE code in [SDW] will make the Project Data Bases unusable. Or the SDW user may be denied access to the data bases due to UIC conflicts.

These potential problems are easily fixed by disallowing the use of the Project Data Bases. This can be done by altering one character in the SDW Top Level module in file "SDWUTIL.COM". The line to be changed in that module is:

```
NOPDBS := "N"
```

This must be changed to:

```
NOPDBS := "Y"
```

in order to disallow the use of the Project Data Bases.

#### 2.4 ALTERING A TOOL'S RESIDENT DEVICE

During the operational period of the SDW, it may be necessary to alter the location of the SDW itself or one of its components. The SDWE was designed to handle this very simply. A logical device name is given to the resident device for each SDW component, as well as, the SDWE itself. These assignments are made in the "SETSDW.COM" file. By simply editing this file so that the logical device assignment is to the new resident device, the tool can be accessed properly from the SDW. This was facilitated by using the logical names for devices within all of the SDW code.

## 2.5 LIMITED DEPTH OF SDWE MODULE CALLING

When using DCL command modules, the software developer is limited to a depth of eight calls. The SDWE itself only uses a maximum of four levels of calling depth and all component tools are called from only a second or third level of depth module. However, if the component tool uses too many further levels of command module calls, a serious limitation is realized. The only manners in which to remedy this problem are to reduce the levels of calls in the component tool or in the SDWE.

## 2.6 SPECIAL LIMITATIONS IN THE SDWE

In order to enforce the information hiding of SDW command external to the current SDW module, the symbols for the SDW commands are assigned and deassigned at the beginning and conclusion of each module. To do this the "EXIT" command given from the command level had to be assign to the value: "GOTO OUT" so that the SDW codes get deassigned properly. However, this disables the use of the EXIT command in modification to any of the SDW modules. Be aware of this fact if you need to modify you copy of the SDW.

### 3. CONCLUSION

The Software Development Workbench and the Software Development Workbench Executive are very young and developing software systems. Your experiences and comments on these systems are greatly appreciated. If you do have any comments or problems with the SDW or SDWE, please forward them to:

Lt. Steven Hadfield, USAF  
P.O. Box 4143  
or  
Dr. Gary Lamont  
Electical Engineering Department

Air Force Institute of Technology  
School of Engineering  
Wright-Patterson AFB, OH 45433

(513) 255-5533

## VITA

Steven M. Hadfield was born on 6 March 1959 in Milwaukee, Wisconsin. He moved to Clearwater, Florida at the age of 1 and graduated from Clearwater Central Catholic High School in 1977. He attended Tulane University on an Air Force Reserve Officers Training Corps (AFROTC) Four-Year Scholarship. He served as Cadet Corps Commander during his senior year. He graduated from Tulane University in 1981 with a Bachelor of Science Degree with majors in both Mathematics and Economics. He was also a Distinguished Graduate of AFROTC. Second Lieutenant Hadfield entered the School of Engineering, Air Force Institute of Technology, in June of 1981.

Permanent address : 12449 84th Way N.  
Largo, Florida 33542

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/EE/82-D-17	2. GOVT ACCESSION NO. AD-A124822	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  AN INTERACTIVE AND AUTOMATED SOFTWARE DEVELOPMENT ENVIRONMENT		5. TYPE OF REPORT & PERIOD COVERED  MS THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)  Steven M. Hadfield, 2nd Lt. USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS  Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS  Computer Integrated Manufacturing Branch Materials Laboratory, Wright Aeronautical Lab. Wright-Patterson AFB, Ohio 45433		12. REPORT DATE  December 1982
		13. NUMBER OF PAGES  376
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)  Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES  Approved for public release; LAW AFR 190-17. <i>LYN E. WOLAVER</i> LYN E. WOLAVER Dean for Research and Professional Development Air Force Institute of Technology (AIC) Wright-Patterson AFB OH 45433		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Computer Software                      Software Development Software Development Environment      Automated Tools Software Engineering                    Software Development Tools Programming Environment                Computer Aided Design (CAD) Integrated Computer-Aided Manufacturing (ICAM)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  The purpose of this investigation is to 1) define both the detailed requirements and the preliminary design for an automated and interactive software development environment, and 2) develop an initial implementation of that environment. The specified requirements for this environment emphasize the need to support the entire software life-cycle as a continuous and iterative process. In particular, the concepts of integration, traceability, flexibility, and user-friendliness are		

20. (con't)

accentuated. The preliminary design delineates the high level design specifications, configuration schemes, and generic tool categories with which the previously mentioned requirements may be satisfied.

Detailed designs are developed for the integrating interface/controller sub-system and the development data storage scheme for the initial implementation of the environment. The interface/controller sub-system has been implemented and tested using the DEC Command Language (DCL) and PASCAL. This sub-system is integrated with an initial software development tool set executing on the VAX-11/780 computer using the VMS operating system. This initial implementation, called the Software Development Workbench (SDW), is an extremely effective and easy to use aid for extending the cognitive and notational powers of the software developer.

END

FILMED

3-83

DTIC